(2)

SOFTWARE MEASUREMENT GUIDEBOOK

MDA972-92-J-1018

SPC-92116-CMC

VERSION 01.00.00

JUNE 1991

SOFTWARE MEASUREMENT GUIDEBOOK

SPC-92116-CMC

VERSION 01.00.00

JUNE 1991

Reprinted for the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

February 1993

SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

SLIM is a trademark of Quantitative Software Management Inc.

CONTENTS

ACKNOWLEDGEMENTS xiii

1. INTRODUCTION 1-1

1.1 Objective 1-1

1.2 Audience 1-2

1.3 Guidebook Content 1-4

1.4 Guidebook Organization 1-4

1.5 Benefits 1-5

1.6 Approach 1-6

1.7 Quick Reference Estimation Guide 1-9

1.8 Summary of Recommendations 1-10

1.9 Typographic Conventions 1-11

2. THE SYSTEM AND SOFTWARE LIFE CYCLE 2-1

3. SETTING QUANTIFIABLE GOALS AND MANAGING TO THEM 3-1

3.1 Managing Within the Size/Schedule/Quality/Cost Envelope 3-1

3.1.1 Size/Schedule/Quality/Cost Envelope 3-1

3.1.2 How to Apply the Quantitative Management Process 3-1

3.1.3 How to Use Measurements in the Quantitative Management Process 3-2

3.2 Managing According to the Goal-Question-Metric Paradigm 3-4

3.3 Quantitative Product Requirements and Process Objectives 3-5

3.3.1 Identifying Critical Requirements 3-5

3.3.2 Quantifying Requirements and Setting Measurable/Testable Targets 3-6

3.3.3 How to Quantify Requirements for Software 3-7
3.3.4 How to Specify Attributes of Requirements 3-8
3.3.5 Unstated Critical Product Attributes 3-9
3.4 How to Benefit From Negotiating Requirements 3-9
3.4.1 How to Negotiate Product Requirements 3-9
3.4.2 How to Benefit From Prior Examples of Quantitative Critical Attributes ... 3-10
3.4.3 Examples of Quantitative Critical Attributes 3-12
3.5 How to Move Colleagues to Quantify Requirements 3-19
3.6 Summary of Recommendations 3-19
4. METRICS DATA DEFINITION AND COLLECTION 4-1
4.1 Metrics and Process Maturity 4-1
4.1.1 Measurements and Indicators 4-1
4.1.2 Data Collection and Process Maturity Levels 4-1
4.2 Measurements and Metrics Definitions 4-2
4.3 Derivation of Labor Rates for New and Reused Code 4-3
4.4 Product Size Metrics 4-5
4.5 Cost Metrics 4-6
4.5.1 Labor Cost Metrics 4-6
4.5.2 Computer Usage Cost Metrics 4-7
4.6 Productivities and Labor Rates 4-8
4.7 Quality Measures and Metrics 4-9
4.8 Schedule Measures 4-10
4.9 The Enterprise Software Experience Database 4-10
4.9.1 The Enterprise 4-10
4.9.2 Minimum Data Set 4-11
4.9.3 Measurements Collection 4-12
4.10 Summary of Recommendations 4-13

5. HOW TO ESTIMATE SOFTWARE SYSTEM SIZE 5-1

5.1 Size Estimation 5-1

5.1.1 Size Estimation Activities 5-1

5.1.2 Size Estimation and the Development Life Cycle 5-2

5.1.3 Size Estimation and Process Maturity Levels 5-2

5.2 Size Estimation During the Development Life Cycle 5-3

5.2.1 Size Estimation by Development Phase 5-3

5.2.2 Using Source Lines of Design to Estimate Software Size 5-4

5.2.3 Size Estimation Steps 5-4

5.3 Function Block Counting 5-4

5.4 Statistical Size Estimation 5-6

5.5 Function Points 5-7

5.6 How to Estimate Software Size by Counting Externals 5-9

5.7 Software Product Size Growth 5-10

5.8 Summary of Recommendations 5-11

6. HOW TO ESTIMATE SOFTWARE COST 6-1

6.1 Overview 6-1

6.1.1 Cost Estimation Methods 6-1

6.1.2 Cost Estimation and Process Maturity Levels 6-1

6.1.3 Units of Cost 6-2

6.2 Holistic Models 6-2

6.2.1 The Constructive Cost Model 6-3

6.2.1.1 Basic Constructive Cost Model 6-3

6.2.1.2 Intermediate Constructive Cost Model 6-4

6.2.1.3 Detailed Constructive Cost Model 6-5

6.2.1.4 Reuse 6-5

6.2.1.5 Ada Process Model 6-6

6.2.2 The Software Life-Cycle Model 6-8
6.2.3 The Cooperative Programming Model 6-9
6.2.4 How to Apply Holistic Models 6-9
6.3 Activity-Based Models 6-10
6.3.1 Activity-Based Model and Process Maturity Levels 6-10
6.3.2 The General Form of the Activity-Based Model 6-10
6.3.3 The Waterfall Model 6-14
6.3.4 The Evolutionary Spiral Model 6-14
6.3.5 Pooling Estimates 6-16
6.3.6 Point and Interval Estimates of Labor Rates 6-17
6.3.7 The Cost Effect of a Higher Order Language 6-18
6.3.8 The Cost Effect of Software Product Size 6-18
6.4 How to Estimate Documentation Costs 6-18
6.5 How to Estimate Testing Costs 6-20
6.6 How to Cross-Check Estimates 6-21
6.6.1 Design Cost Estimate Example 6-21
6.6.2 Testing Cost Estimate Example 6-21
6.7 Top-Down Estimation of Total System Development Costs 6-21
6.8 How to Estimate Costs of Support to Software Development 6-23
6.9 Risk in Estimates of Cost 6-24
6.9.1 Point Estimates of Cost 6-24
6.9.2 Interval Estimates of Cost 6-25
6.10 Organizational Considerations 6-28
6.11 Software Maintenance Costs 6-29
6.12 Summary of Recommendations 6-29
7. HOW TO ESTIMATE SCHEDULE 7-1
7.1 Schedule Estimation Overview 7-1
7.2 Estimating the Development Schedule 7-2

7.3 Schedule Impact of Reused Code 7-3
7.4 Schedule/Development Effort Tradeoff 7-4
7.5 Schedule/Effort/Size Compatibility 7-4
7.6 Software Development Labor Profiles 7-5
7.7 Summary of Recommendations 7-7
8. ESTIMATING DEFECT CONTENT 8-1
9. MANAGEMENT INDICATORS FOR TRACKING AND MONITORING 9-1
9.1 Management by Measurement 9-1
9.1.1 Software Development Project Tracking and Monitoring 9-1
9.1.2 Project Surveillance and Process Maturity Level 9-2
9.2 Management Indicators 9-2
9.3 How to Compute Management Indicators 9-5
9.3.1 Software Product Size Indicators 9-5
9.3.2 Software Cost Indicators 9-5
9.3.3 Development Schedule Indicators 9-6
9.3.4 Project Technical Stability Indicators 9-6
9.3.5 Project Status Indicators 9-6
9.3.6 Quality Indicators 9-6
9.3.7 The Product Completion Indicator 9-6
9.3.8 Computer Resources Indicators 9-9
9.4 Data for Software Development Project Surveillance 9-10
9.4.1 Data Sources 9-10
9.4.2 Surveillance Activities 9-11
9.4.3 Data Collection 9-11
9.5 The Estimate at Completion 9-13
9.6 Graphical Methods of Monitoring and Control 9-14
9.7 Summary of Recommendations 9-17

10. PROCESS MATURITY HIERARCHY: FRAMEWORK FOR MEASUREMENT 10-1

10.1 Overview 10-1

10.2 Process Maturity Levels 10-2

10.2.1 Process Maturity Level One 10-3

10.2.2 Process Maturity Level Two 10-3

10.2.3 Process Maturity Level Three 10-4

10.2.4 Process Maturity Level Four 10-4

10.2.5 Process Maturity Level Five 10-4

10.2.6 Summary of Measurement-Related Activities 10-4

10.3 Quantitative Management Functions by Process Maturity Level 10-5

10.4 Measurement Activities in Process Maturity Hierarchy 10-7

10.4.1 Software Engineering Institute Assessment Focus on Process Characteristics 10-7

10.4.2 Software Engineering Institute Assessment Questions by Process Characteristics 10-9

10.4.2.1 Measurement 10-9

10.4.2.2 Tracking 10-9

10.4.2.3 Error Analysis 10-9

10.4.2.4 Process Analysis 10-9

10.4.3 Organizational Characteristics by Process Maturity Level 10-10

10.4.3.1 Level Two 10-10

10.4.3.2 Levels Three and Higher 10-12

10.5 Measurement Support 10-12

10.6 Summary of Recommendations 10-17

GLOSSARY Glos-1

REFERENCES Ref-1

BIBLIOGRAPHY Bib-1

QUESTIONNAIRE Ques-1

FIGURES

Figure 1-1. Quantitative Software Management Process 1-7

Figure 3-1. Quantitative Software Management Process 3-3

Figure 3-2. Requirements Hierarchy 3-7

Figure 6-1. Cumulative Distribution of Costs 6-27

Figure 7-1. Schedule Compatibility Testing Process 7-5

Figure 7-2. Development Effort Planning Curve 7-6

Figure 9-1. The Monitoring and Control Process 9-11

Figure 9-2. Example of Monitoring Defects Per Thousand Source Lines of Code by Review or Inspection 9-14

Figure 9-3. Example of Status Tracking 9-15

Figure 9-4. Patterns of Program Trouble Report Opening 9-15

Figure 9-5. Patterns of Program Trouble Report Closure 9-16

Figure 9-6. Example of Defect Density Monitoring and Control 9-16

Figure 9-7. Example of Computer Resource Monitoring and Control 9-17

Figure 10-1. Measurement Foundation for Software Engineering Institute Levels Three Through Five 10-1

Figure 10-2. Fishbone Chart for Attaining Process Maturity Level Two 10-19

Figure 10-3. Fishbone Chart for Attaining Process Maturity Level Three 10-20

Figure 10-4. Fishbone Chart for Attaining Process Maturity Level Four 10-21

Figure 10-5. Fishbone Chart for Attaining Process Maturity Level Five 10-22

TABLES

Table 1-1. Advancing the Software Development State of Practice 1-2

Table 1-2. Functional Responsibility and Related Guidebook Sections 1-2

Table 1-3. Quick Reference Estimate Guide 1-9

Table 1-4. Measurement-Related Activities by Process Maturity Level 1-11

Table 3-1. Tradeoffs Among Software Cost, Size, Schedule, and Quality 3-1

Table 3-2. Checklist: Basic Steps for Successful Software Project Management 3-2

Table 3-3. Examples of Goal-Question-Metric Paradigm 3-5

Table 3-4. Example of Quantifying Functional Objectives 3-7

Table 3-5. Examples of Critical Requirements 3-8

Table 3-6. Example of Performance Objective, Minimum Acceptable, and Current Levels of Requirements 3-9

Table 3-7. Examples of Critical Attributes for Software Products 3-11

Table 3-8. Attribute Specification Template 3-13

Table 3-9. Example of Attribute Specification Format Template for Quantified Attribute 3-15

Table 3-10. Example of Attribute Specification Format Template for a Functional Attribute 3-16

Table 3-11. Software Quality Factors 3-16

Table 3-12. Quality Attributes and Appropriate Measures 3-18

Table 4-1. Software Experience Database Measurements Collection Form 4-14

Table 5-1. Measurement-Related Activities by Process Maturity Level 5-1

Table 5-2. Methods for Estimating Software Size 5-3

Table 5-3. Size Estimation Table Example 5-7

Table 5-4. Function Count Weights for Complexity 5-8

Table 5-5. Code Growth Factors 5-11

Table 5-6. Recommendations on Methods for Estimating Software Size 5-11

Table 6-1. Basic Constructive Cost Model Effort and Schedule Equations 6-3

Table 6-2. Embedded Mode Activity and Schedule Distribution 6-4

Table 6-3. Intermediate Model Effort Multipliers 6-4

Table 6-4. Adaptive Quantities by Phase for Equivalent Delivered Source Instructions ... 6-5

Table 6-5. Weights for the Ada-Constructive Cost Model 6-7

Table 6-6. Sample Effort and Productivities for the Ada-Constructive Cost Model 6-8

Table 6-7. Worksheet Cost Calculations for an Activity-Based Model 6-13

Table 6-8. Waterfall Model Activities and Labor Rates 6-15

Table 6-9. Example of Evolutionary Spiral Model Activities and Labor Rates 6-17

Table 6-10. Estimating Pages From Software System Size 6-19

Table 6-11. Estimating Documentation Effort From Document Size 6-19

Table 6-12. Estimating System and Software Testing Effort 6-20

Table 6-13. Top-Down System Development Estimating Models 6-22

Table 6-14. Top-Down Software System Cost Estimating Example 6-23

Table 6-15. Percent Additional Cost for Support to Software Development 6-24

Table 6-16. Example of Size and Unit Cost Risk 6-25

Table 6-17. Example of Derivation of Distribution of Costs 6-26

Table 6-18. Example of Distribution of Costs 6 27

Table 6-19. Recommendations on Methods for Estimating Software Costs 6-29

Table 7-1. Inferred Technology Constant Values 7-2

Table 9-1. Software Management Indicators and Metrics 9-3

Table 9-2. Example of Product Completion Indicator Calculation 9-8

Table 9-3. Data Sources for Software Management Indicators 9-10

Table 9-4. Management Indicators Work Measurements Collection Form 9-12

Table 9-5. Management Indicators Quality Measurements Collection Form 9-13

Table 10-1. Levels of Software Process Maturity 10-2

Table 10-2. Measurement-Related Activities by Process Maturity Level 10-5

Table 10-3. Management Activities by Process Maturity Level 10-6

Table 10-4. Measurement Foundations of Software Engineering Institute Process Maturity Level Hierarchy 10-8

Table 10-5. Organizational Characteristics of Software Engineering Institute Process Maturity Level Hierarchy 10-11

Table 10-6. Software Management Indicators and Metrics 10-12

ACKNOWLEDGEMENTS

The authors of this guidebook are Robert Cruickshank, John Gaffney, and Richard Werling. The Consortium wishes to thank David Weiss for his support during the writing of this guidebook and Charlotte Winnick for her great efforts in getting this guidebook published. Ron Damer, Dave Owens, Tim Powell, Kent Johnson, Jerry Decker, Sam Redwine, and Charlotte Winnick provided helpful criticisms and suggestions in reviewing the material. Stephanie White, Andy Rabinowitz, Tom Bowen, and Milt Waxman of the Consortium member companies also provided insightful comments.

This page intentionally left blank.

1. INTRODUCTION

1.1 OBJECTIVE

The *Software Measurement Guidebook*[1] provides a set of quantitative management methods designed to help you improve your management of large software projects. The methods presented, which are based on experience on such projects, have proven effective and inexpensive. The guidebook presents state-of-the-art material, much of which is also state-of-practice in various development organizations. It supports the objectives of quantitative management: establishment of goals, prediction/estimation of product and process, and assessment to support project and process control and modification. The guidebook prescribes the measurements that you need to take to satisfy various software management objectives but not the actions that you might take as a result of them.

The guidebook will help meet the DOD Instruction 5000.2 requirement on software metrics:

> Software management indicators and metrics will be used in the management of the software effort and will relate to continuous improvement action using analysis of lessons learned, post-development problems, and quality performance rate and records against pre-established criteria. These indicators and metrics will be described in the Computer Resources Life-Cycle Management Plan (Department of Defense 1991, 6-D-4).

This guidebook is designed to:

- Provide practical methods of quantitative software management.
- Present methods for the quantitative establishment of goals for software process and product, estimation, and tracking. It defines practical metrics and describes how you can obtain and apply them during software development.
- Provide specific guidance to satisfy the measurement needs for attaining each higher level of process maturity as defined by the Software Engineering Institute (*SEI*). This version of the guidebook emphasizes what is required at levels 2 and 3.
- Provide specific measurement guidelines for the waterfall model and the Evolutionary Spiral Model.[2]
- Include lessons learned from experience.
- Serve as (1) standalone, (2) reference on measurement for other Consortium guidebooks, or (3) as a whole or in part with member company software standards.

1. This edition of the guidebook does not contain Sections 2 and 8 and the Index. A later version will contain them.
2. While the waterfall model is subsumed by the Evolutionary Spiral Model, it is treated separately for those who are using only the waterfall model.

The quantitative management methods presented are intended to help you advance your state of practice (in terms of the SEI process maturity levels), as seen in the examples given in Table 1-1. This course of action helps you expand your scope from programming technology to the technology of quantitative software management. It takes advantage of recent methods (systematic inspections and statistical methods for improved control of software development processes) to improve the stability and predictability of software development processes. All groups cited in Table 1-2 should make use of the appropriate guidebook sections.

Table 1-1. Advancing the Software Development State of Practice

Move Beyond SEI Process Maturity Levels 1 and 2	**Move Toward SEI Process Maturity Levels 3 Through 5**
Programming technology	Quantitative software management technology
Rough prediction	Precise prediction and control
Reviews and walk-throughs	Statistical process control methods using systematic inspection methods

Table 1-2. Functional Responsibility and Related Guidebook Sections

Functional Responsibility	**Applicable Section**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Hardware/software system manager, senior managers, organization software development	X									X
Software project manager	X		X				X		X	X
Lead software engineer	X				X	X	X		X	
Software engineer				X	X	X	X			
Cost engineer, measurement analyst, system analyst		X	X	X	X	X	X	X	X	
Software quality engineer				X				X	X	
Proposal manager	X	X	X	X					X	
Financial manager	X					X	X		X	
Financial analyst						X	X		X	

1.2 AUDIENCE

This guidebook addresses the measurement needs of line managers and senior system and software engineers. It explains what points in the software process you are to *measure*; the nature of the data you should track for process and product control; and the relation of the measures to management decisions that you need to make based on them. The guidebook will be useful to a broad spectrum of member company people, particularly those who are concerned with improving the predictability and control of the software process employed and of the software it produces. As shown in Table 1-2, these people include managers, software engineers, and others involved in implementing the software process (systems and software line managers, project managers, business area managers, proposal managers, financial managers, and senior financial analysts).

Functional responsibilities are defined as follows:

- Senior manager: Division or corporate vice president, or equivalent, responsible for authorizing a measurement program across all software projects. Responsibility includes authorizing both direct costs and the indirect (overhead) expense of the measurement program.

- Hardware/software system manager: The person responsible for managing a project containing both hardware and software. He uses the *Software Measurement Guidebook* to cross-check estimates, and to monitor and control the software-based portions of the project.

- Software project manager: The person responsible for managing a software-based project. He uses the *Software Measurement Guidebook* to estimate, monitor, and control the project.

- Lead software engineer: A technical supervisor responsible for the development or support of a software-based system. He supervises the use of prescribed methods to perform technical activities.

- Software engineer: The person responsible for development or support of a software-based system. He uses prescribed methods to perform technical activities.

- Cost engineer, measurement analyst, or system analyst: A technical staff member responsible for collecting project cost and schedule status data and for analyzing and projecting project performance.

- Software quality engineer: A technical staff member responsible for collecting data from design and code reviews and inspections and for analyzing and projecting product quality.

- Proposal manager: The person responsible for describing and supporting the estimated cost, schedule, and quality of a software product.

- Financial manager: The person responsible for consistency in estimating, tracking, and monitoring the cost of software products.

- Financial analyst: The person responsible for tracking and monitoring the cost of software products.

Table 1-2 relates the functional responsibilities of various members of the guidebook audience to the sections which they might be expected to find especially interesting. For example, Sections 1 and 10 address the concerns of managers of hardware/software systems and of corporate- or division-level software development managers. The software project manager will find his concerns covered in Sections 1, 3, 7, 9, and 10.

This guidebook is not meant to be a comprehensive treatise on software measurement. Rather, it is designed to meet the "how to" objectives in Section 1.1. You may find reference to other texts useful when gaining understanding of some of the guidebook material; relevant references are identified. Substantial effort may be required to understand and apply some of the material presented in the guidebook. However, the quantitative techniques used are, with the exception of Section 4.3, at the level of first-year college mathematics.

1.3 GUIDEBOOK CONTENT

The principal subjects covered by the guidebook and the corresponding sections are:

- The nature of the development process and its relationship to the work breakdown structure (*WBS*) for software products (Sections 2 and 3).
- Methods for making credible estimates of size (Section 5), cost (Section 6), schedule (Section 7), and quality (Sections 3 and 8).
- Recommendations for data collection for product and process metrics (Sections 4 and 9).
- Establishment of quantifiable requirements for software products and processes (Section 3) and methods to track their degree of attainment within the cost/schedule/quality envelope throughout the life cycle (Sections 3 and 9).
- Methods for providing measurement support (monitoring, controlling, and predicting) for activity-based models, including the waterfall model and Evolutionary Spiral Model (Sections 4, 6, 7, and 9).
- Methods for providing measurement support specific to reaching higher SEI process maturity levels (Section 10).

Included separately with this guidebook is a quick reference card that helps you find sections that show how to estimate commonly used parameters such as the computer software configuration item (*CSCI*) size and development effort.

You can use parts of this guidebook alone. In particular, you can use Sections 5 (size), 6 (cost), and 7 (schedule) without the other sections.

1.4 GUIDEBOOK ORGANIZATION

The guidebook is composed of ten sections. They are:

- Section 1, Introduction, describes the guidebook's objectives, benefits, and intended audience. This section includes a quick reference estimation guide that lists various functions (such as size estimation) and related formulas and points to guidebook sections having more detail.
- Section 2, The System and Software Life Cycle, describes the system and software life cycle, the waterfall and evolutionary spiral software life cycles, and process management principles using the entry-task-verification-exit (*ETVX*) paradigm (Radice and Phillips 1988). This section is not included in this edition of the guidebook.
- Section 3, Setting Quantifiable Goals and Managing to Them, shows how to establish quantifiable software process and product requirements and monitor the degree of their realization throughout the development process. It includes measures and measurement points consistent with DOD-STD-2167A requirements for both the waterfall model and Evolutionary Spiral Model. Using approaches compatible with those for the Evolutionary Spiral Model, this section also describes how to accommodate changes in requirements throughout the software

life cycle. This section is related to Section 4, Metrics Data Definition and Collection, and Section 9, Management Indicators for Tracking and Monitoring. Together, they deal with improving project management by using metrics, data collection, and review.

- Section 4, Metrics Data Definition and Collection, defines measurements and metrics required to credibly estimate and effectively monitor software size, quality, schedule, productivity, and cost. The section describes project data for an organization's historical software experience database. It gives guidance on data collection.

- Section 5, How to Estimate Software System Size, shows practical ways to estimate software size before development by considering the inputs and outputs, number of computer software units (*CSUs*), computer software components (*CSCs*), CSCIs, and "boxes" in system diagrams and by using statistical techniques.

- Section 6, How to Estimate Software Cost, explains the use of several types of cost models in the life-cycle context using practical approaches. Experience indicates that the activities of data collection and analysis consume less than two percent of project cost. You can more than recapture this cost by reducing the proportion of project effort required for unnecessary rework.

- Section 7, How to Estimate Schedule, describes how to estimate software project schedules and how to do tradeoffs between effort and schedule.

- Section 8, Estimating Defect Content, explains how to provide baselines for reviews and tests by estimating *defect* content. This section is not included in this edition of the guidebook.

- Section 9, Management Indicators for Tracking and Monitoring, supplements Section 4 and recommends a minimum set of status data to be collected and used for project management and control. Working with the status data, the section defines indicators of process, product, and progress to aid timely, corrective action before problems become expensive to fix. The emphasis is on how to predict certain measures such as cost at completion during the software development process. It provides methods that are applicable to software projects using the waterfall model and Evolutionary Spiral Model and gives guidance on data collection for tracking and monitoring and for graphical methods of presentation.

- Section 10, Process Maturity Hierarchy: Framework for Measurement, describes what measurement technology you must use as part of the software process to attain SEI process maturity levels 1 through 5.

In addition, a Quick Reference Estimate Card is provided in the pocket of this guidebook. It is also provided as Section 1.7. A questionnaire about this guidebook is also included. Please make a copy, fill it in, and mail it to the Consortium after you have used the guidebook.

1.5 BENEFITS

This guidebook was developed to help you improve your control of the software process by applying quantitative software management techniques. It was created to help fulfill the Consortium's long-term goal to help you more quickly attain the benefits of higher quality software at a lower net cost. The guidebook presents methods for predicting and monitoring the software process and the software products it generates. These methods are consistent with principles of the SEI process

maturity concept of software management technology. Implementation of the methodology is designed to shorten the elapsed time required to attain higher SEI process maturity levels, producing higher quality, more usable software products. In addition, the guidebook provides specific measurement guidance for implementing the measurement aspects of the Evolutionary Spiral Model and of progression to higher levels of (SEI) process maturity.

1.6 APPROACH

Figure 1-1 summarizes the process that the guidebook recommends. The guidebook provides examples and templates to aid you in applying the methods presented.

Cost estimation is a fundamental part of the cost management process. As shown in Figure 1-1, cost management begins at the preproposal and proposal stages, before the development process starts, when you establish and verify customer and user requirements. Once the software development activity has begun, you review and revise cost performance plans and goals as necessary. You monitor the development project for cost problems through the life cycle, especially at major milestones. At each milestone, you make cost estimates and compare them with the planned goals. At the end of software development, you should collect actual project cost data and place it in your enterprise's software cost and size database for use in future estimates. You should perform cost estimation within a process of cost management that you continuously apply throughout the entire development. (Quinnan 1980) and (Boehm 1981) discuss the steps within the estimating and management process.

The steps in the software project cost estimation and management process (Figure 1-1) are:

- ***Verify Your Understanding of the Software Requirements and Goals.*** This must begin in the preproposal and proposal stages. It is important for you to verify the completeness of your team's understanding of the stated and unstated users' needs and requirements, both domain and unique, and of the ultimate application environment. The specifications' correspondence with operational requirements often is incomplete or distorted. Completing this step reduces the probability of omissions in architecture, design, and coding processes.

- ***Plan Project Work.*** Given this understanding, plan the project work.

  - Use system requirements and preliminary system design to make initial estimates of software size, schedule, and quality requirements (see Sections 5, 6, and 7). After you have determined targets for size, quality, and schedule, use historical cost data to establish cost plans and objectives.

  - Design a WBS that shows all software process activities and captures their costs by activity, by CSC and CSCI, and by language if required. Ensure that the development schedules and manpower plan are consistent with the product size and cost estimates and with the software cost plans and objectives.

  - Perform cost and function tradeoffs.

  - Estimate the cost risk as a function of the difference between your final estimate and the proposed price. Report this estimated risk to management (see Section 6.9).

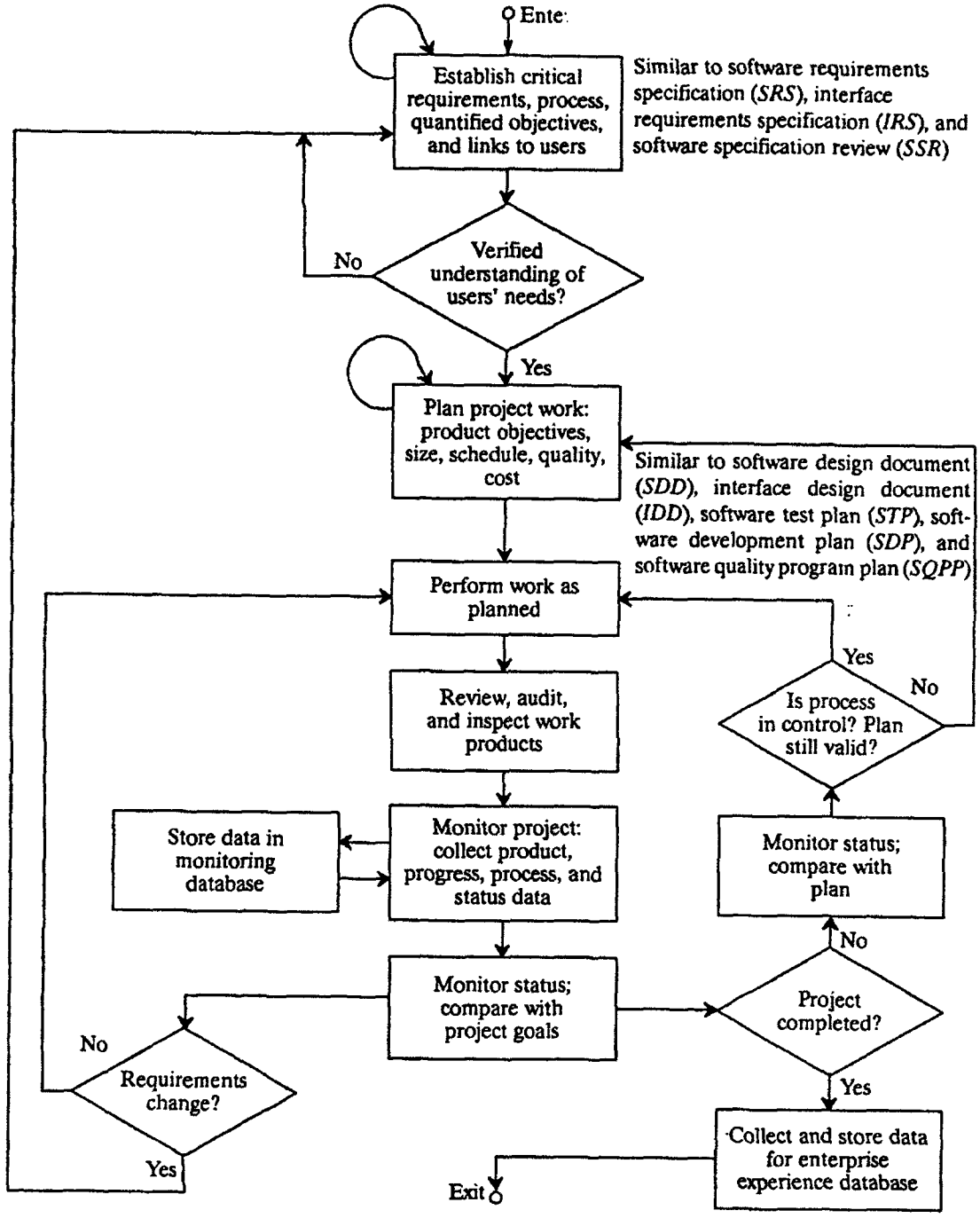


Figure 1-1. Quantitative Software Management Process

- At the initiation of development, plan cost performance. The plan includes defining the points in time (e.g., milestones or specified time intervals) at which you will reestimate the cost as well as the data collection procedures, metrics, and project management indicators to be used. Your organization's software process standards will provide guidance on the data to be collected and metrics to be used (see Sections 4 through 9).

  - Make cost proposal estimates using several different techniques, and reconcile them among themselves and with any other pertinent estimates (see Sections 4, 5, 6.2 through 6.5, 6.8, 6.11, 7, and 8).

- ***Monitor Project Progress.*** During the development project, monitor project performance by collecting data on product status, project progress, and process performance. Use this data to project estimates of product size, schedule, quality, and overall cost at the planned points in time (see Section 9 for definition and discussion). Compare these projections to those planned and budgeted.

  Revise project plans and objectives as required by the variance of the actual from the planned progress.

  Revise project plans and objectives as you modify requirements (see Section 9).

- ***Store Project Data for the Enterprise Software Experience Database.*** At the end of the project, collect actual costs and sizes and enter them into your enterprise's (organization's) software experience database (see Sections 4 and 10).

The decisions you make at a software project's start are the hardest to change because of their cumulative effects on later stages of the process. Consequently, you should base those decisions on the knowledge that the project team correctly understands the users' critical requirements (i.e., those which must be met for the system to satisfy user needs). Base all subsequent project plans and budgets on a few critical performance requirements formalized in the early stages of development. Unfortunately, you must often change even the best of those plans and the budgets derived from them in response to modifications made to the system's requirements and an increased understanding of the system's behavior obtained during the development process.

You should describe requirements in quantitative terms and associate them with testable and measurable targets that enable you to verify performance objectively. There are two types of requirements: qualities or benefits and resources (Gilb 1988). The former relates to the product, its reliability, and ease of use. The latters relate to the constraints (people, time, and money) that are limits on attaining the benefits desired by the product. The critical requirements of each category include:

- Qualities/benefits: Functions that the software is to perform, or what the software is to do, and the environment in which to implement and execute the designated functions. Both the technical and business aspects of the environment are covered. How well the software must work (i.e., the quality of the software product) is part of this requirement.

- Resources:

  - The size of the software to be developed. Virtually all estimates of cost are based on the estimated size of the software products. Four indicators that you can determine early in the project (number of external inputs, number of external outputs, number of external inquiries, and number of interfaces with other programs in the system) help in estimating this.

  - The reliability, complexity, and run-time constraints of the target computer needed by users.

- Schedule or work time to required delivery.
- The price to the user (not necessarily the same as cost).

1.7 QUICK REFERENCE ESTIMATION GUIDE

Table 1-3 is a quick reference estimation guide. It summarizes how to estimate certain key items such as development effort.

Table 1-3. Quick Reference Estimate Guide

Estimate Of	Point in Process	Input Required	Formula	Output	Section
Software system size	Project initial stages	C_1 = Number of CSCIs	$S = 41.6C_1$	*KSLOC*	5.3
Software system size	Project initial stages	A = Sum of 3 externals	$S = 13.94 + 0.034A$	KSLOC	5.6
		E = Sum of 4 externals + interfaces	$S = 12.28 + 0.030E$	KSLOC	
CSCI size	Project initial stages	C_2 = Number of CSCs	$S = 4.16C_2$	KSLOC	5.3
Development effort, *COCOMO*	Any (when size is known or estimated)	1,000 delivered source instructions (KDSI)	$LM = a(KDSI)^b$ for organic, semidetached, or embedded modes	LM	6.2.1
Development effort, *COPMO*	Any (when size is known or estimated)	S = 1,000 lines of source code (KSLOC) L = Average staffing level in LM/month	$E = a + bS + cL^d$ for COPMO model	LM	6.2.3
Unit cost, development effort	Any (when size is known or estimated)	L/K, unit cost in LM/KSLOC for each activity, KSLOC	Total Cost = $\Sigma(L/K)_i$KSLOC waterfall model or Evolutionary Spiral Model	LM	6.3
Size, effort, or development time, given any two (Putnam)	Any (when size is known or estimated)	C, technology constant S, size in KSLOC K, effort, labor years t_d, development time in years	$S = CK^p t_d^q$	KSLOC and/or labor years and/or years	6.2.2, 7.2
Schedule, COCOMO	Any	LM = labor months TDEV = development time in months	$TDEV = c(LM)^d$ for organic, semidetached, or embedded modes	months	6.2.1
Schedule/effort tradeoff	Any	K_0, estimated effort t_0, estimated schedule K_1, new estimated effort t_1, new estimated schedule	$K_1 = K_0\left(\frac{t_0}{t_1}\right)^{q/p}$	labor years	7.4
Scheduled labor profiles	Any	K, estimated effort t_d, estimated schedule		labor months per month	7.6
Document pages	Any	KSLOC estimate	$P = a(KSLOC) - b(KSLOC)^2$	pages	6.4
Documentation effort	Any	KP = thousand pages	LM = uKP, *LH* = vKP	LM (or LH)	6.4
Testing costs	Any	R = Number of testers T = Number of test steps	$LH = (a + bR)T$	LH	6.5

Table 1-3, continued

Estimate Of	Point in Process	Input Required	Formula	Output	Section
Problem correction costs	Any set of problems resulting from a test phase	T = Number of test procedure steps	LH = cT	LH	6.5
Top-down estimation of total project costs	Preproposal or proposal stages	Software development total unit costs in LM/KSLOC and size in KSLOC	Table 6-13 or previous experience data	LM	6.7
Costs of support to software development	Software development planning	Total software development costs	Cost for support in each area = a • (software development cost) where a is a multiplier less than 1. See Table 6-15.	Any	6.8
Risk estimate of cost	Any	Knowledge of distribution of size and cost estimates	Point and interval estimates of risk	Probability and LM	6.9
Software maintenance	Any	Size in KSLOC	Cost = (defects/KSLOC) • KSLOC • (LM/defect)	LM	6.11

1.8 SUMMARY OF RECOMMENDATIONS

Software development organizations should try and, as appropriate, adapt the quantitative software management methods presented in this guidebook. You should customize the methods, formulas, and parameter values to your own organization's operating characteristics and experience. Implementing this recommendation helps your organization shift toward an advanced emphasis on the technology of engineering and management for the entire software life cycle, a change from the traditional emphasis on controlling programming technology. Termed the quantitative software management process, Table 1-1 summarizes this approach.

You should adapt and implement, as appropriate, methods presented for predicting and monitoring your software process and the software products it generates. As shown in Table 1-4, the methods are consistent with the principles of the SEI process maturity concept of software management technology. Implementing the methodology will aid you in achieving higher SEI process maturity levels, allowing you to produce higher quality, more usable software products and simultaneously accelerating improvements in your software development process.

Your software development organization should base its measurement program on its present level of process maturity. Tables and fishbone charts in Section 10 present guidance to estimate present levels of process maturity, and they indicate the actions needed to reach the next level. These tables and charts are designed to simplify this recommendation.

The primary benefit to member companies of having a sound measurement program is increasing the degree of software process and product predictability and control. This benefit is derived from more and better information, which makes better management decisions possible. This can be called "management by measurement."

Table 1-4. Measurement-Related Activities by Process Maturity Level

Level 2, Repeatable Process	Level 3, Defined	Levels 4 and 5
Estimate, plan and measure software size, resource estimates, staffing levels, schedules, and development cost. Maintain profiles over time for units designed; build/release content; units completing test; units integrated; and test progress, requirements changes, and staffing. Maintain profiles over time for utilization of target system memory, throughput, and input/output (*I/O*) channels. Collect statistics on design *errors* and on software code and test errors found in reviews and inspections.	Level 2 data, plus: • Maintain formal records for progress of unit development. • Maintain formal records for test coverage.	Levels 2 and 3 data, plus: • Routinely project, compare to actuals, and analyze errors found in reviews and inspections of requirements, designs, and code and test. • Routinely analyze software productivity for major process steps. • Maintain a managed and controlled process database for process metrics across all projects.

1.9 TYPOGRAPHIC CONVENTIONS

This guidebook uses the following typographic conventions:

Serif font ..... General presentation of information.

Italicized serif font ..... Words, expressions, abbreviations, and acronyms found in the Glossary, mathematical expressions, and publication titles.

Boldfaced serif font ..... Section headings and emphasis.

This page intentionally left blank.

2. THE SYSTEM AND SOFTWARE LIFE CYCLE

This page intentionally left blank.

3. SETTING QUANTIFIABLE GOALS AND MANAGING TO THEM

This section presents a method of quantitative software management built around the concept that effective management requires effective measurement. It describes how to express requirements in quantitative terms and associate them with testable/measurable targets that enable you to verify performance objectively. The method is adapted for, but not restricted to, embedded real-time software. Section 3.1 is a top-level view of managing within the envelope of size/schedule/quality/cost constraints. Section 3.2 shows you how to derive a minimum set of measurements needed to manage a software project effectively using the goal-question-metric, or *GQM*, paradigm (Weiss 1981; Basili and Weiss 1984). The project's goals drive the minimum set of management data. These goals lead to project management questions that you can answer by analyzing project data. The questions in turn lead to the set of metrics needed for effective management of a project. Section 3.3 shows you how to establish quantitative product requirements and process goals.

3.1 MANAGING WITHIN THE SIZE/SCHEDULE/QUALITY/COST ENVELOPE

3.1.1 Size/Schedule/Quality/Cost Envelope

The process of planning software development projects must work within the envelope of constraints defined by a software product's cost, size, schedule, and quality. You can optimize any one of these four—if you can ignore the other three (although such luxuries are not often found in the real software engineering world). In practice, project managers make constant tradeoffs among these variables. An attempt to optimize any one constraint often impairs or suboptimizes the other three. When project cost is limited artificially, product schedule and quality will probably suffer, and the resulting product may lack certain functions. As shown in Table 3-1, whenever you change one constraint, the other three inevitably change, too. In summary, skilled project management is the ability to maintain an acceptable balance among such conflicting requirements.

Table 3-1. Tradeoffs Among Software Cost, Size, Schedule, and Quality

Optimizing a Project's	Can Force Suboptimization Of
Scheduled completion date	Cost, quality, size
Cost	Schedule, quality, size
Quality of product	Schedule, cost
Size	Cost, schedule, quality

3.1.2 How to Apply the Quantitative Management Process

Table 3-2, which is synthesized from (Gilb 1988) and (Humphrey 1988), is a top-level view of the vital steps in managing software development projects.

Table 3-2. Checklist: Basic Steps for Successful Software Project Management

Step	Basic Steps for Successful Management
1	Before planning the work, assess the completeness of your team's understanding of both the stated and unstated user needs and requirements in the ultimate application environment. This seemingly trivial reminder is key to managing evolving requirements. Its importance has been demonstrated often, yet the step is seldom actually completed. For example, minimum levels of critical operational requirements (i.e., the subset of software requirements that the software must meet for the system to satisfy user needs) are often not identified as such.
2	Plan the work. First, agree in advance on quantified measures of success such as product performance (minimum and objective levels), quality, size, cost, schedule, and project risk. Agree on how success will be recognized, including responsibility for acceptance testing, criteria to be used, and any consequences of *failure*. Establish continuing links with customers who know the application well, to stay aware of approaching changes in requirements. Four related positive suggestions for this step can be helpful:
2a	Plan, whenever possible, to provide an early delivery of some immediately useful results.
2b	Include a planned and easy retreat path for each step in case your team oversteps its abilities.
2c	Divide the work into independent parts. Define precisely the requirements for each part.
2d	Commit to your work and work to meet your commitments.
3	Control relationships among the parts that comprise the WBS, CSCs, and CSCIs.
4	Perform project work as planned, capitalizing on your project team's experience, inspiration, and motivation to develop products that fully satisfy user needs.
5	Review, audit, and inspect intermediate products at each stage of software development to ensure that the work is done as planned and that project objectives are likely to be met.
6	Routinely collect project status data for product, progress, and process. Store periodic status data in a project monitoring database.
7	Remember that software development is a learning process; recognize what you do not know. When the gap between your knowledge and the task is severe, fix the gap before proceeding.
8	Monitor project status and compare it with the plan. Monitor formally at software requirements review (*SRR*), system design review (*SDR*), preliminary design review (*PDR*), critical design review (*CDR*), and similar reviews, and monitor routinely according to your organization's internal policies.
9	Track and maintain the plan against project goals. When requirements change, you probably need to revise the plan. Use management indicators for "early warning."
10	Refine the plan periodically during the project as your knowledge of the project improves.
11	At project completion, collect and store the final data in an enterprise experience database.

This top-level list, while simplistic, is designed to emphasize steps in which management attention to quantification is especially productive. Coincidentally, the checklist serves as a reminder of hard-learned lessons that often are forgotten or overlooked under the pressures of starting a project.

3.1.3 How to Use Measurements in the Quantitative Management Process

Figure 3-1 is a graphic overview of the quantitative management process, showing the generation and use of measurement data that takes place during the development process. Table 3-2 outlines the steps in this process that you should follow for successfully managing your software project. The goals and budgets you establish at step 2 (perhaps modified as requirements evolve) serve as reference points

for actual status during the project. You monitor the actual project status in step 8, where you compare project achievement to date against planned achievement to date.

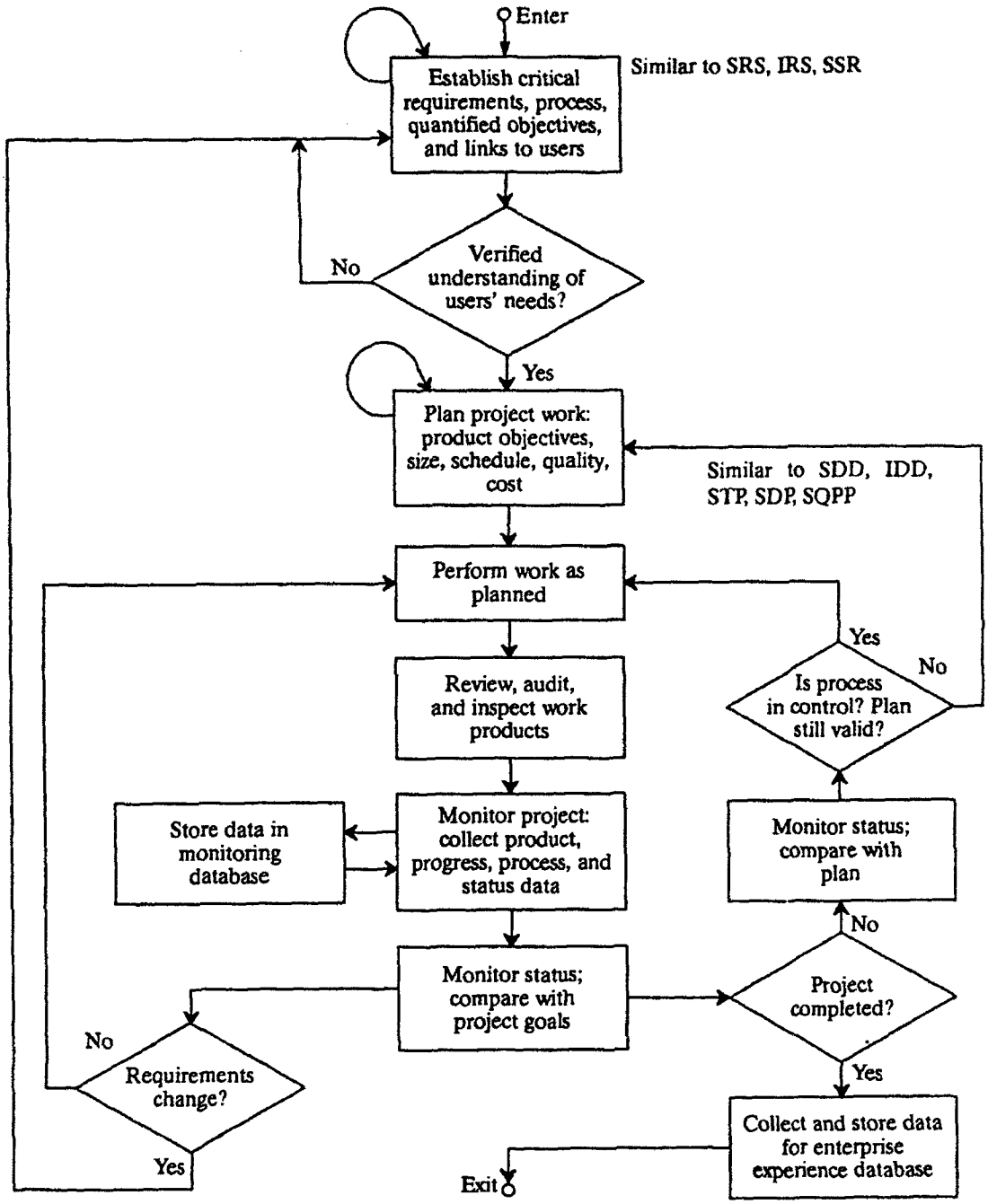


Figure 3-1. Quantitative Software Management Process

The decisions you make throughout the project are guided by the anticipated effects on the few critical requirements. Much unnecessary project rework is caused by insufficient attention to the first two project steps, which are so crucial to project success.

Step 1 in Table 3-2 is a reminder that, in spite of great care in preparation, the correspondence of specifications with evolving operational requirements is often incomplete and distorted. The first project step is represented by the decision diamond, Verified understanding of users' needs? It confirms that: (a) users believe their needs for critical product performance are understood and expressed quantitatively, in terms that will allow the degree of achievement to be tested and measured; (b) critical product requirements have been set (including the critical requirements and project objectives that seemed so obvious); and (c) the users have agreed to them. Finally, it verifies that links with knowledgeable user(s) have been established to quickly become aware of changes in user needs or product requirements as the project proceeds. In DOD-STD-2167A terms, this step includes the work of the SRS, IRS, and the functional baseline established after completion of the SSR.

Step 2 in Table 3-2 suggests that you plan the project work within the constraints of the size/schedule/quality/cost envelope, as described in Section 3.1.1. A useful tactic, for which an investment in time will be repaid as the project proceeds, is to deliver some products of high value to the user(s) at early steps during the project. In terms from DOD-STD-2167A, products of this step include preliminary SDD and IDD and the STP. Management-oriented products include equivalents to the SDP, SQPP, and STP. It is useful to revisit the decision point Verified understanding of users' needs? and get user sign-off after you complete your planning.

This guidebook focuses on collecting, analyzing, and monitoring status data on product, progress, and process (steps 5, 6, and 9). How to perform the planned work is not discussed further here. That is subject to the procedures of a given organization.

3.2 MANAGING ACCORDING TO THE GOAL-QUESTION-METRIC PARADIGM

You should select software process and product measurements according to the GQM paradigm (Weiss 1981; Basili and Weiss 1984). GQM is a framework for the systematic specification of measures appropriate for some need that you identify. Use of the paradigm helps you avoid picking measures "out of the air." Following it aids you in identifying what measures you should use based on a rationale for their selection. The paradigm has three parts:

1. State your goal. This answers the question,What do I want to know or do?

2. State the item or items of information you want to know. This answers the question, What questions am I going to ask to satisfy the goal?

3. State the specific things you need to measure, directly or indirectly, so that you can answer each of the questions that you have posed in step 2 of the quantitative management process. This step answers the question, What do I need to measure (directly or indirectly, calculated from one or more direct measures of the thing or process of interest to you)?

The power of the GQM paradigm is that it helps you reduce the costs of data collection by helping you concentrate on a minimum set of metrics, each of which is clearly shown to support project goals such as being able to assess the likelihood of attaining some product development objective by staying within a maximum cost bound. The metrics resulting from the application of the GQM paradigm quantify the characteristics of software products, processes, and development progress that are most useful to project management (Gilb 1988). Table 3-3 shows an example of the logical progression from goal to metric for goals of cost, schedule, and system performance.

Table 3-3. Examples of Goal-Question-Metric Paradigm

System Goals	Project Management Question	Resulting Metric
Meet cost objective within agreed range	How much has the project work cost during the most recent time interval? What will be the cost at completion?	Labor hours and dollars per unit (CSCI, module) in most recent time interval Projected cost at present rate of progress, extended to completion
Meet schedule objective within agreed range	At present rate, will the project finish by the scheduled date? How much of the planned work has been completed to date?	Projected completion rate based on current cumulative progress rate Proportion of software units (CSCI, modules) designed, coded, unit tested, and integrated to date
Meet system performance objectives within agreed ranges (e.g., for execution rates, workload processed, quality, ease of adapting to changed parameters)	What is the current estimate of system performance relative to project objectives?	Results of simulations or performance analysis Perceived ease of adaptation on a scale of one to ten

For example, the **goal** of this guidebook is to help a project team get higher quality software at lower net cost and net elapsed time by improving its project management methods. The **question** follows from the goal, e.g., how do you use quantitative management technology in planning, organization, control, and technical leadership activities? The relevant **measure**(s), then, are those needed to answer that question.

3.3 QUANTITATIVE PRODUCT REQUIREMENTS AND PROCESS OBJECTIVES

The preceding sections stress the importance of clearly identified and quantifiable functional requirements, quality objectives, and resource limitations to project success. This section describes how to develop critical product requirements on a software project and how to express them in measurable/testable terms. This technique of quantifying requirements brings to project teams significant power to control their efforts. Part of the power rests in this fact: requirements are not discovered; they evolve and are negotiated. That is, they result from a discussion between the software person verifying the requirements and the customer for the system.

3.3.1 Identifying Critical Requirements

This section describes how to identify the small number of "critical attributes" so essential in determining how well the end user wants the software product to work in its intended operational environment. For mission-critical software, you usually draw these from performance, reliability, user satisfaction, and maintainability. You should not expect software to be accepted if its performance level for a measurable/testable critical attribute is less than the minimum acceptable level specified by the users. In identifying the critical requirements for software, you should also consider the relevance of system-specific critical system characteristics mentioned in *Department of Defense Instruction 5000.2*, Defense Acquisition Management Policies and Procedures (Department of Defense 1991). These may include survivability, transportability, electronic countermeasures, energy efficiency, and interoperability, standardization, and compatibility with other forces and systems including post-deployment support organizations.

Requirements are often "shopping lists." They are nearly always much too long to use as practical management tools. To cope with this problem, managers identify the handful of "critical" requirements—those few that the software must meet (within ranges prespecified with users) for the system to satisfy users' minimum needs—and concentrate on them. The Pareto distribution (20 percent of the variables account for about 80 percent of the effort and cost) characterizes project requirements. Impacts on the few critical requirements will exert inordinate influence in guiding project management decisions throughout the project. Their very importance requires that you identify and carefully confirm them. Projects which fail to specify their objectives clearly, and fail to exercise control over even one single critical attribute, can expect project failure to be caused by that one critical attribute (Gilb 1988).

The recommended solution for how to express, in measurable/testable terms, "how well" the system should work is to quantify the requirements of the attributes of the system qualities or benefits. You cannot complete the first two blocks in Figure 3-1 until this is done. Gilb insists that this is almost always possible, although he acknowledges that there is "a certain art to finding the necessary metrics concepts and measuring tools. Often they have no traditional written form, and it is essential that they are tailored to the case in hand" (Gilb 1988, 35).

3.3.2 Quantifying Requirements and Setting Measurable/Testable Targets

This section describes how to express requirements in quantitative terms. You need to know how much user need for each requirement is to be met to deliver software products that meet user operational requirements on time and within budget. Software projects are funded to provide operational benefits to users, and the system benefits are usually expressed in terms of prespecified performance requirements. Thus it is reasonable that successful software projects begin with user concerns and with the benefits they anticipate from a completed system. Agreement is aided when you describe requirements in quantitative terms and associate them with measurable/testable targets that enable you to verify performance objectively.

The greatest difficulty in establishing quantitative requirements is in getting a "clear, complete, unambiguous statement of our quality requirements" (Gilb 1988, 32). Do this by recognizing two different types of attributes: true/false and multiple value. True/false attributes are requirements with only one future state: true or untrue. For example, a project will use Ada for all new code, or it will not. In contrast, multiple value attributes can be expressed on scales of measurement. You normally express space and time objectives, such as cost and schedule requirements, and measure them in quantitative terms. However, you can describe objectives for user-required functions such as work capacity and adaptability, which are seldom expressed quantitatively, in measurable/testable terms; for example, "at the specified workload, in 99.9 percent of transactions, response time will be less than 1.3 milliseconds." As the project progresses, you can verify the likely satisfaction of the objective at each phase (e.g., low-level design) by static analysis and/or simulation and modeling tools.

An example may help make these distinctions clear. The project requirements might include performance objectives such as those in the first column of Table 3-4. Attributes could be quantified using metrics defined as shown in Table 3-4. Quantifying a requirement begins by identifying the functional objectives needed for success. The "critical" requirements for project success are those considered essential by the user. In this example, the critical requirements include execution/workload processing rates and quality. You can quantify each of these so that its attainment becomes measurable and testable (Gilb 1988).

Table 3-4. Example of Quantifying Functional Objectives

Functional Objectives (Meet system performance objectives for:)	**Quantified Attributes of Objective** (How well it must be done)
Execution rates	Under prespecified conditions and using the specified data sample, the CSCI executes at rates no less than 1,300 transactions/second.
Workload processed	Within specified conditions, the system processes at least 130 transactions/second. At specified degraded capability, it processes at least 72 transactions/second.
Quality	Transaction accuracy is no less than 99.5 percent under all conditions.

In summary, you can quantify and test both functional and attribute objectives, and you can measure the degree to which the system's performance meets these objectives.

3.3.3 How to Quantify Requirements for Software

Quantifying requirements enables you to establish clear, complete, quantified, and agreed-upon (by developer and intended users) objectives for software products and processes. Quantified requirements must be measurable/testable statements of the results users really want and statements of the costs they are willing to incur for those benefits. The quantitative objectives you establish for each product goal (especially for software quality characteristics) permit you to make meaningful assessments of project risk based on probabilistic analysis at each stage of the software development project. Figure 3-2 is an overview of quantifiable requirements.

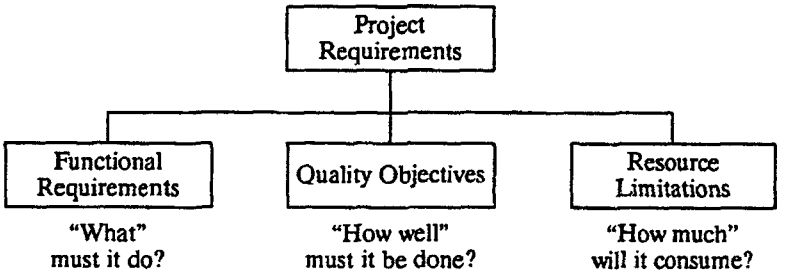


Figure 3-2. Requirements Hierarchy

The three categories of requirements are:

- ***Functional Requirements.*** These represent the functions to be performed by the software: what the software is to do. The extent and nature of functions the software must perform for the system drives the size of the software product.

- ***Quality Objectives.*** Quality may be defined as conformance to requirements (Gaffney 1981). In the case of software, how well users want the software to work specifies the required quality of the software product. You can and should state all quality requirements unambiguously. Ease of use, reliability, maintainability, and portability are typical requirements that you can quantify (Gilb 1988). Examples of user quality objectives are ultra-high reliability for a communications processing system and *fault* tolerance, safety, and robustness for a nuclear weapon system.

- ***Resource Limitations.*** Resource limitations identify such constraints as dollar cost, labor hours, computer time, and calendar months available until delivery.

  The number of work hours is the primary driver of cost and, to a lesser extent, other resource expenses incurred in development activities (such as computer support). Note that the price the user pays is not always the same as the cost the developer incurs.

3.3.4 How to Specify Attributes of Requirements

Decisions made at the beginning of a software project, and therefore hardest to change, are the most vital ones. Base your subsequent project plans and budgets on the few critical performance requirements that you formalize in this early work. As those plans and budgets change in response to the availability of more information about the system's behavior and to the concurrent evolution of requirements, a quantitative baseline is essential for evaluating effects of changes on schedule and cost.

Table 3-5 shows how you can express negotiated requirements in quantitative terms. The Critical Performance Requirements columns list what the product must do and how well it must do it, while the third column identifies the method for measuring performance.

Table 3-5. Examples of Critical Requirements

Examples: Critical Performance Requirements		
What it Must Do	**How Well**	**Method of Measurement**
Pass acceptance tests for critical abilities (by physical test or simulation)	Specified in terms that can be measured and tested	Specify agreed-on test or measuring tool
Transaction rate at given volume levels	Transactions/second	Transactions/second, under simulated field operating conditions
Response time for given transaction loads	Less than 0.3 seconds from receipt of incoming signal	Simulation prior to final qualification test (*FQT*)
Availability, etc.	More than 99.8 percent	FQT
Pass Design Review	Fewer than 10 major faults to be corrected before proceeding	Pass review, with no open major faults from CDR

You must measure each performance requirement on some scale and verify it by some test. You will find it helpful to make comparisons among current level of performance, minimum acceptable level (at delivery) and performance objective level (at delivery), and the "best ever," or current state of the art. Reference the authority that quantitatively specifies the minimum acceptable and performance objective levels.

In the example shown in Table 3-6, the first column shows the four levels of performance that need to be quantified for each of two typical requirements: response time and availability. The Critical Performance Requirements columns list what the product must do and how well it must do it. The performance objective and minimum acceptable levels are those on which the users and software developers have agreed. The best ever (record, state of the art, or best performance known) puts these levels into perspective by comparing the performance objective and minimum acceptable levels with the state of the art. The current level shows performance of whatever current system is in use, whether technology is manual or automated.

Table 3-6. Example of Performance Objective, Minimum Acceptable, and Current Levels of Requirements

Performance Level	Critical Performance Requirements			
	What it Must Do	How Well	What it Must Do	How Well
Performance objective level (at delivery)	Response time	0.1 second	Availability	99.8%
Minimum acceptable	Response time	0.5 second	Availability	99.2%
"Best ever" level	Response time	0.01 second	Availability	99.9%
Current level	Response time	1.1 second	Availability	98.5%

3.3.5 Unstated Critical Product Attributes

Be alert for an additional danger: the critical attributes that are never mentioned explicitly. "Obvious" things which "everybody knows" cannot be left to take care of themselves. When you read a statement of management objectives, you should assume that several of the most important attributes have not been stated (Gilb 1988, 39). This, paradoxically, is often because these objectives are **so essential** that the customer takes them for granted. Work with your customer to assure yourself that these hidden critical requirements are identified and stated clearly.

3.4 HOW TO BENEFIT FROM NEGOTIATING REQUIREMENTS

Experienced developers have learned that all other project activities depend on getting agreement on which of the final end user's needs the software must fulfill (what the user needs), the extent to which those different needs are to be fulfilled (the quality), and the resource limitations.

Because requirements are normally incomplete and involve tradeoffs, setting measurable/testable targets involves negotiation. It is worthwhile. Quantitative terms permit agreement to be better communicated so that both the user and developer can measure product performance. Recognizing that requirements changes are not free, at each refinement of requirements you should also refine the corresponding estimates of size, schedule, and resource projections (see Sections 5, 6, and 7).

3.4.1 How to Negotiate Product Requirements

The term "negotiating" is useful for describing the process frequently observed during the course of a project. Negotiating is a search for "win-win" alternatives, by which no one loses. Negotiating is an interactive process that converges upon a cost-effective set of requirements. The negotiation process is cyclical and continuous throughout a project. Initial requirements are negotiated prior to, and during, the SRR milestone. Changes occur after PDR as users make tradeoffs among costs and desired product functionality.

You should follow these steps when negotiating requirements:

1. Identify the subset of critical product attributes that **must** be met for the product to survive and be successful. These attributes are the means for determining project success, and you need to control them throughout the project. Other product attributes are typically those in the "nice to have" category (Humphrey 1988, 88). When possible, use customer language.

2. Differentiate critical attributes of requirements from feasible solution(s), which are **alternative** ways of obtaining the results you want. Solutions are often confused with objectives, particularly in the early stages. There is real danger in developing solutions to meet end user needs before you have broken down those needs into product "attributes" and quantified and agreed to those attributes. You must differentiate between customer requirements and feasible solutions, because a feasible solution can so easily be mistaken for a requirement (Gilb 1988, 48-53).

3. Quantify the minimum acceptable level for each critical attribute, e.g., software quality. This is a crucial step. Often in the past the "minimum" level of performance and the "performance objective" have not been differentiated clearly. In Part 4B2c, DOD Instruction 5000.2 clearly intends that the distinction be made in future acquisitions (Department of Defense 1991). This step addresses that intention.

4. Find and describe critical attributes that may seem so obvious that they are not written down. When you read a statement of customer objectives, it is wise to assume that several of the most important attributes have not been stated. This, paradoxically, is often because these objectives are so critical that the customer takes them for granted (Gilb 1988).

5. Identify all product attributes in terms of measurable/testable results needed by the final end user. Attributes can vary within some limits, and the system will still be acceptable to the user.

6. Establish a continuing link with someone who knows the end user's application to assist in predicting and accommodating change in requirements throughout the life cycle. If possible, the link should be someone who was involved in the earliest negotiations for testable/measurable product attributes.

7. Revise requirements and requirement attributes as quickly as possible, when appropriate. The world is dynamic; its dynamism affects requirements and solution feasibility. Even in the case of incremental improvements to an existing product, requirements will change.

3.4.2 How to Benefit From Prior Examples of Quantitative Critical Attributes

You would be wise to negotiate the specific measures, with the quantified measurable/testable attributes expressed in the customer's language. A substantial body of literature is available to help you in the quantification process. Table 3-7 shows examples of common critical attributes for software products that you can quantify. It shows examples of both critical performance factors and limited resources. The two columns are independent of each other. The left-hand column shows how you can express the critical performance requirements of workability, availability, adaptability, and usability in quantitative terms. The table shows each attribute with a brief definition and an {example measure}. For instance, here are the definition and examples for process capacity:

> Process capacity: measure of ability to process units of work in units of time. {measure chosen: units of useful work per unit of time: transactions/second, display refreshes/second}.

Equally necessary, and easier to quantify, are the limited development resources such as calendar time, people, and money shown in the right-hand column.

Table 3-7. Examples of Critical Attributes for Software Products

Examples of Project-Critical Attributes	
Qualities Critical to Success	**Limited Resources**
Workability: Measure of the raw ability of the system to perform work. Subattributes include: • Process capacity: Measure of ability to process units of work in units of time. {measure chosen: units of useful work per unit of time: transactions/second, display refreshes/second}. • Responsiveness: Measure of reaction to a single event. {measure time between two events in an "ask-answer" process: time from question understood by operator until answer displayed on screen (no load), response time (when worst case activity load exists)}. • Storage capacity: The capacity of the system to store units of any defined thing. {Characters/record, source instructions}.	Time: Subattributes include: • Calendar time elapsed to build a system. • Work days needed to accomplish a task.
Availability: Measure of how much a system is usefully available to perform the work which it was designed to do. Subattributes include: • Reliability: Measure of the degree to which the system does what it is intended to do, as opposed to something else. {Measure mean time between failures (*MTBF*), Mean time to next failure (*MTNF*)}. • Maintainability/improvability: Measure of how quickly an unreliable system can be brought to a reliable state. In general, includes recovery from the effects of a fault. {Measure time needed to correct errors, possibly artificially inserted}. • Integrity: Measure of the trustworthiness of the system. Is it in the state it is supposed to be in? {Simulate threats of different types to measure ability of installed security techniques to counteract them}.	People: Covers all people-related resources such as "work years" to construct a system and the people needed to staff or operate it. Serves as limiting objectives when designing a system and when controlling its resource consumption in operation.
Adaptability: Measures the system's ability to change or be changed without uncontrolled side effects. Subattributes include: • Improvability: {Negotiate, express in customer's language}. • Extendability: Measure of ease of expansion of data storage requirements or computational functions. {Negotiate, express in customer's language}. • Portability: {Negotiate, express in customer's language}.	Money: Covers all types of the monetary costs of building and maintaining the system.

Table 3-7, continued

Examples of Project-Critical Attributes	
Qualities Critical to Success	**Limited Resources**
Usability: Measure of how well the intended group of people will be able and motivated to use the system productively. {Negotiate, express in customer's language. Measure time needed for people of given qualification to learn to use a particular system well enough to produce results at a certain rate, opinion surveys, employee turnover rates}. Subattributes include: • Entry requirement. • Learning requirement. • Handling ability. • Likability.	Tools: Physical resources which are critical to project success and which can somehow be limited. {Negotiate, express in customer's language}. Subattributes include: • Computer capacity. • Office space. • Rights to use software packages and designs.
Other qualities: {Negotiate, express in customer's language}. Subattributes include: • Accuracy: Measure of precision required in calculations, outputs. • Error tolerance: Measure of continuity of operation under non-nominal conditions. • Simplicity.	Other resource: Subattributes include: • Marketing costs. • Goodwill, reputation, image.

3.4.3 Examples of Quantitative Critical Attributes

You must measure each performance requirement on some scale and verify it by some test. You will find it helpful to make comparisons among current level of performance, minimum acceptable level (at delivery) and performance objective level (at delivery), and the "best ever," or current state of the art. Table 3-8 is a useful format with which to start. The left column lists essential data items. The right column documents specific values for the software product to be developed by your project.

The template items are:

- Name of Attribute: Identifies the system and subsystem (e.g., a CSCI). It includes the date prepared, author, and approval authority to make it easier to identify the latest version. A short acronym or abbreviation for the attribute description may simplify communication about it.
- Requirement: Identifies the specific critical requirement for which this template quantifies attributes.
- Scale: Specifies the measure to be used, qualified by any set of conditions which must hold true for this specification to apply.

- Date: Specifies the date or event at which the attribute will be measured or tested.

- Test: Identifies a practical test or measuring tool which must be used to determine whether you are within performance objective and minimum limits.

Table 3-8. Attribute Specification Template

Attribute Specification Format Template	
Item	**Quantified/Functional Attribute**
Date Author System Subsystem Approved by: Name of Attribute:	
Requirement:	
Scale =	
Date =	
Test =	
Minimum =	
Performance Objective =	
Record =	
Now =	
Reference:	

- Minimum: Identifies the minimum acceptable limit on the scale for the measurable/testable attribute. A performance result of measurement/test that is worse than this minimum level is defined as total system failure, no matter how good other attributes may be. Note that the distinction between "minimum" and "performance objective" agrees with DOD Instruction 5000.2, 4B2c.

- Performance Objective: Specifies the level of performance on the Scale, expected by the Date at which the attribute will be measured or tested. Often, contractors do not distinguish this level from the "minimum" acceptable level.

- Record: Specifies an engineering limit, state-of-the-art limit, or best ever example expressed in quantitative terms. It is used for global comparison, and is not an expected requirement in this system.

- Now: Gives quantitative performance data for some presently existing system for comparison with the performance objective and minimum acceptable levels. This current level shows the performance of whatever current system is in use, whether technology is manual or automated. It is not a requirement.

- Reference: Specifies the authority that quantitatively specifies the minimum acceptable and performance objective levels. It also provides reference to more detail or other related documents.

The performance objective and the minimum acceptable levels are those on which the users and software developers have agreed. The best ever (record, state of the art, or best performance known) puts these levels into perspective by comparing the performance objective and minimum acceptable levels with the state of the art. More detail on templates can be found in (Gilb 1988, 359–389).

Following are examples of measurable/testable critical attributes typical of those required for software in embedded systems. These examples, which illustrate the idea in tabular form, are adapted from (Gilb 1988, 7–8, 360–361). Table 3-9 provides an example of an attribute specification format, an attribute that you can measure on a continuous scale.

The second example, Table 3-10, shows how you quantify an attribute when it has only two states, satisfactory or not satisfactory. The example chosen is compliance with DOD Instruction 5000.2, Section 4-C, Critical System Characteristics. In testing compliance, each requirement will be rated as true or false; either it meets the requirement or it does not.

Quality factor ratings have been developed for some years. One study, adapted in Table 3-11, describes 13 quality metrics developed for acquisition purposes (Bowen, Wigle, and Tsai 1985). Table 3-11 identifies additional metrics that may be useful for quantifying certain internal performance requirements.

You may find it useful to investigate known measures that you may be able to apply. Table 3-12 shows a selection from the many measures that have been developed and proposed for quantifying quality attributes. The table, which is not exhaustive, was adapted from (Watts 1987, 18, 26; Hoecker et al. 1984). Detailed information about the various quality measures in the table is available in these references. Only entries that have been demonstrated in commercial or industrial situations are listed. Those limited to laboratory demonstrations have been omitted from the list.

Table 3-9. Example of Attribute Specification Format Template for Quantified Attribute

Attribute Specification Format Template	
Item	**Quantified/Functional Attribute**
Date Author System Subsystem Approved by: Name of Attribute:	 Work capacity
Requirement: Work capacity of software.	Practical work capacity must be sufficient to handle the prespecified workload mix within the prespecified hardware processing capacity.
Scale = {scale of measure specification} The qualifier may be any set of conditions which must hold true for this specification to apply.	Average time per transaction at workload level of 10,000 transactions/second. (Attribute requirements are expressed on scales of measurement).
Date = {the date or event specified; the date or event to which the other specifications apply when no qualifier is specified}	Formal test or simulation to be completed before stated milestone.
Test = {a practical test or measuring tool which must be used to determine whether you are within performance objective and minimum limits}	Simulation, using a prespecified sample of transaction data, to be performed on the specified system.
Minimum = {minimum acceptable limit on the scale; worse than this is defined as total system failure, no matter how good other attributes are}	Minimum acceptable case (for trial use): 10 000 transactions/second at a prespecified workload mix.
Performance Objective = {the level on the Scale expected by the Date}	Performance objective level (initial real use): 14,000 transactions/second.
Record = {engineering limit, state-of-the-art limit, or best ever; used for global comparison and is not an expected requirement in this system}	Record: 1,800,000 transactions/second, set 1/23/89 on ABC system. (For global comparison, cite performance of comparable systems, measurable in the same way. The record performance will usually exceed the level of performance objective).
Now = {some presently existing system for comparison with the performance objective and minimum acceptable levels; not a requirement}	Existing system, which software is to replace, handles only 1,800 transactions/second.
Reference: {reference to more detail, authorization, or other related documents}	User document(s) where agreement on requirements levels, measurement procedures, etc., is recorded.

Table 3-10. Example of Attribute Specification Format Template for a Functional Attribute

Attribute Specification Format Template	
Item	**Quantified/Functional Attribute**
Date Author System Subsystem Approved by: Name of Attribute:	Compliance with DOD Instruction 5000.2, Section 4-C, Critical System Characteristics.
Requirement: Compliance With DOD Instruction 5000.2, Section 4-C, Critical System Characteristics.	Software design features critical to successful operational and support consider [true/false]: survivability; transportability; electronic counter-counter measures; energy efficiency; and interoperability, standardization, and compatibility with other forces and systems including support infrastructure.
Scale = {scale of measure specification} The qualifier may be any set of conditions which must hold true for this specification to apply.	Functional requirement with only one future state: true or untrue.
Date = {the date or event specified}	Will be verified at CDR.
Test = {a practical test or measuring tool which must be used to determine whether you are within performance objective and minimum limits}	Critical design documentation will be compared with DOD Instruction 5000.2, Section 4-C, Critical System Characteristics.
Minimum = {minimum acceptable limit on the scale; worse than this is defined as total system failure, no matter how good other attributes are}	Minimum acceptable case: Meets all requirements (approved waivers rated acceptable) except for electronic counter-countermeasures.
Performance Objective = {the level on the Scale expected by the Date}	Performance objective: Meets all requirements (approved waivers rated acceptable).
Record = {engineering limit, state-of-the-art limit, or best ever; used for global comparison and is not an expected requirement in this system}	Name of another comparable system that is in complete compliance.
Now = {some presently existing system for comparison with the performance objective and minimum acceptable levels; not a requirement}	Performance of existing system which software is to replace.
Reference: {reference to more detail, authorization or other related documents}	Identify SDD, IDD, and STP along with any approved waivers.

Table 3-11. Software Quality Factors

Acquisition Concern	User Concern	Quality Factor Definition	Possible Metric
Performance: How well does it function?	How well does it use a resource?	Efficiency: Relative extent to which a resource is used (e.g., storage space, processing time, I/O time).	Actual resource utilization / Allocated resource utilization
	How secure is it?	Integrity: Extent to which the software will perform without failures due to unauthorized access to the code or data within a specified time period.	Errors / 1,000 hours operation simulated under prespecified conditions

Table 3-11, continued

Acquisition Concern	User Concern	Quality Factor Definition	Possible Metric
	What confidence can be placed in what it does?	Reliability: Extent to which the software will perform without any failures within a specified time period.	Errors / 1,000 hours operation simulated under prespecified conditions
	How well will it perform under adverse conditions?	Survivability: Extent to which the software will perform and support critical functions without failures within a specified time period when a portion of the system is inoperable.	Errors / 1,000 hours operation simulated under prespecified adverse conditions
	How easy is it to use?	Usability: Relative effort for using software (training and operation, e.g., familiarization, input preparation, execution, output interpretation).	Labor days to learn to use / Labor years to develop tested under prespecified conditions
Design: How valid is the design?	How well does it conform to the requirements?	Correctness: Extent to which the software conforms to its specifications and standards.	Traceability demonstration
	How easy is it to repair?	Maintainability: Ease of effort for locating and fixing a software failure within a specified time period.	Average labor days to fix simulated under prespecified conditions
	How easy is it to verify its performance?	Verifiability: Relative effort to verify the specified software operation and performance.	Labor hours to verify, using prespecified conditions
Adaptation: How adaptable is it?	How easy is it to expand or upgrade its capability or performance?	Expandability: Relative effort to increase the software capability or performance by enhancing current functions or by adding new functions or data.	Effort to expand / Effort to develop simulated under prespecified conditions
	How easy is it to change?	Flexibility: Ease of effort for changing the software missions, functions, or data to satisfy other requirements.	Average labor days to change simulated under prespecified conditions
	How easy is it to interface with another system?	Interoperability: Relative effort to couple the software of one system to the software of another system.	Effort to couple / Effort to develop simulated under prespecified conditions
	How easy is it to transport?	Portability: Relative effort to transport the software for use in another environment (hardware, configuration, and/or software system environment).	Effort to transport / Effort to develop simulated under prespecified conditions
	How easy is it to convert for use in another application?	Reusability: Relative effort to convert a software component for use in another application.	Effort to convert / Effort to develop simulated under prespecified conditions

Table 3-12. Quality Attributes and Appropriate Measures

Use	Characteristic	Quality Attribute	Quality Measures
Product operation	Usability	Operability Training/familiarization Communicativeness I/O volume and rate Data commonality Rejection rate by customers Customer complaint rate	**Usability** Flesch-Kincaid readability index Fog index Keystroke-level model Maier's measure of user friendliness Percent of sales returned Complaint rate, percent of sales
Product operation	Security (integrity)	Control and audit of access Integrity of data, etc.	**Integrity** Svan's integrity metrics
Product operation	Efficiency	Efficiency of storage Execution	Efficiency (none)
Product operation	Correctness	Completeness Traceability Consistency	**Correctness** Error prediction measure Model of defect removal Pimont-Rault's testing measure
Product operation	Reliability	Accuracy Error Tolerance Simplicity Consistency	Reliability Complexity measure based on selection and nesting Gilb's logical complexity measure Knot count complexity measure Minimal intersection number Myers' interval complexity measure Thayer's logic complexity measure 2-tuple complexity measure Execution time measure Nelson's reliability Numerical reliability Schick/Wolverton's probabilistic measure Littlewood's measure of availability
Product transition	Portability	Self-descriptiveness Modularity Machine independence Software system independence	Gilb's portability measure
Product revision	Maintainability (adaptability)	Consistency Simplicity Conciseness Self-descriptiveness Modularity	De Young/Kampen's readability measure Gordon's program clarity measure Joergensen's readability Halstead's measure E McCabe's cyclomatic number McClure's complexity measure Oveido's program complexity measure Yin/Winchester's measure of software design quality Zolnowski/Simmons' complexity measure Logical stability measure Gilb's structural complexity measure Keutgen's measure of intramodular structure; Maekawa's measure of extensibility Myers' module independence measure Mohanty/Adamowicz's test Paige's test effort Pimont/Rault's testing measure Test effectiveness metric Yin's independence metric

3.5 HOW TO MOVE COLLEAGUES TO QUANTIFY REQUIREMENTS

You need to motivate both your colleagues and your customers to quantify requirements. These approaches have succeeded in allaying concerns and fears (Gilb 1988, 392–401):

- Convince project management to provide **more reward** for trying to put a number on things rather than penalties for not providing perfect numbers.
- Appeal to your colleagues'/customers' **need for clarity**. Quantifying requirements will reduce the risk of a proposal failing because of misunderstood requirements. It will stimulate an earlier, more intense discussion among customer personnel about what they really want. As a result, you can prepare better estimates based on firmer understanding of the real requirements. Your company may use "design by objectives" to systematically engineer and cost a solution to the customer's requirements.
- **Relate or trace** quantified software requirements to customer and management requirements. By doing so, the customer is much more certain of getting what he wants. At the same time, the customer cannot later surprise you with more demanding results requirements than you expected to have to deliver **without** being willing to pay for the changes.
- Show that quantified requirements do **not** require exact knowledge. Rough estimates (that you can refine as data become available) are more useful than none at all, even when your colleagues do not know precisely. Often, you must express estimates for controversial or especially significant values as uncertainty estimates, expected values, or ranges of possible values. For further protection against misinterpretation, attach a short description of the assumed conditions under which the estimate is valid.
- Show your colleagues that quantified requirements are **not** set in stone. As user needs change (they always do), your company can more easily show that those needs are different from **earlier approved** statements and justify a cost change.

3.6 SUMMARY OF RECOMMENDATIONS

By using the recommendations presented in this section as a foundation, you can benefit fully from the detailed measurement techniques presented in the following sections.

- Reduce the amount of unnecessary rework by completing the first basic step in Table 3-2 (assess the completeness of your team's understanding of user needs/requirements, and plan the work).
- In spite of the inevitable pressures, do not short-change the second basic step in Table 3-2 (plan the work) in the rush to proceed. You need to agree on how success will be recognized (including responsibility for acceptance testing), the criteria to be used, and any warranties or other consequences of failure. You should clearly establish links with customers who know the application well (to stay aware of approaching changes in requirements).
- Select measurements of software process and product according to the GQM paradigm. Using this paradigm helps you identify what measures you should use based on a rationale for their selection. The paradigm provides a systematic approach to metric selection. It has three parts:
  - Your goal answers the question, What do I want to know or do?

- The question answers, What control information do I need to know?
- The metric identifies the measures needed to answer the questions posed.

The power of the GQM paradigm is that it helps reduce the costs of data collection by helping you concentrate on a minimum set of metrics, each of which clearly supports project goals.

- Identify and monitor the handful of critical requirements, the few that the software must meet (within ranges prespecified by users) if the system is to satisfy minimum user needs. The critical attributes typically include measures describing how well the end user wants the software product to work, such as performance, reliability, user satisfaction, maintainability, and extendability. No user will accept software if its performance level for a critical attribute is less than the worst acceptable level he specified.

- Have measurements relate to requirements, and express requirements in quantitative terms. Knowledge of how much user need for each requirement is to be met is indispensable to delivering software products that meet user performance requirements and that are delivered on time and within budget. Select the measures based on users' concerns and with the benefits they anticipate from a completed system. Future agreement is aided when you describe requirements in quantitative terms and associate them with testable/measurable targets that enable you to verify performance objectively.

4. METRICS DATA DEFINITION AND COLLECTION

4.1 METRICS AND PROCESS MATURITY

4.1.1 Measurements and Indicators

This section defines a set of software process and product measurements for size, cost, productivity, quality, and schedule that you can adopt to define, control, and improve your software development process. It lists quantities that you can measure and use to calculate metrics which give insight into software products and software development processes. You may wish to collect, compute, and analyze the full set or a selected subset, depending on organizational needs, abilities, and objectives.

You should select software process and product measurements according to the GQM paradigm (Weiss 1981; Basili and Weiss 1984). This paradigm is a framework for the systematic specification of measures appropriate for some need that you identify. Use of the paradigm helps you avoid picking measures "out of the air." Following it aids you in identifying what measures you should use based on a rationale for their selection. The paradigm has three parts:

1. State your goal. This answers the question, What do I want to know or do?
2. State the item or items of information you want to know. This answers the question, What question am I going to ask to satisfy the goal?
3. State the specific thing you need to measure, directly or indirectly, to answer each of the questions that you have posed in step 2. This step answers the question, What do I need to measure (directly or indirectly calculated from one or more direct measures of the thing or process of interest to you)?

This section discusses data collection for a software experience database based on the cost, size, and schedule performance of the software project at completion. Section 9 discusses the collection of product, process, and progress measurements and metrics data during software development for project tracking and monitoring of ongoing projects. These two data collection efforts are for different purposes, and it is important to understand the distinction between the two types of data collection effort here. Section 4 discusses measurements that you should collect at the completion of software development for inclusion in the historical software experience database. You should use these measurements, representing actual software development experience, not only for evaluating overall project performance, but also for setting future software development performance and control standards.

4.1.2 Data Collection and Process Maturity Levels

(Humphrey and Sweet 1987) present a set of data items that you should collect and analyze to improve and optimize your software development process. This set is discussed in the context of achieving SEI

process maturity levels 2 through 5. Many of the SEI data items overlap the set of indicators presented in Section 9 of this guidebook, and most of the data items in common are associated with achieving process maturity level 2 or 3. Section 10 of this guidebook correlates the SEI process maturity level data item requirements with the indicators listed in Section 9 and with the methods presented in Sections 3 through 8 of this guidebook.

You should begin data collection as soon as you decide to establish a metrics program. At process maturity levels 1 and 2, your organization will at first concentrate on collecting software product cost and size data, as in Sections 4.4, 4.5, and 4.6, since you can immediately apply this data. At levels 3 and 4, your organization will add process data such as the quality data listed in Section 4.7.

You may begin data collection at the lower process maturity levels using the minimum data set as described in Section 4.9.2, and continue at the higher process maturity levels with an augmented data set.

4.2 MEASUREMENTS AND METRICS DEFINITIONS

Software measurables are directly observable quantities that you can count, such as source statements (source lines of code [*SLOC*]) or that you can otherwise measure, such as labor hours (LH) and labor months (LM). A measurable is a primitive. A software measurement is a number assigned to a quantifiable aspect (i.e., a quantitative assessment) of a software system (DeMarco 1982).

A software metric is a number assigned to a quantifiable concept that relates to a software product or to the process that created it. A metric is not always directly observable. A metric may be a single measurable as defined above, or it may be a function of one or more measurables. An SLOC is a measurable that is an indicator of software system size; therefore, it is also a metric. SLOC/LM is a productivity metric that is composed of two measurables.

A new software system is composed of new code and reused code. Reused code comes from an "original" system from which you can remove code and modify it for reuse, remove and reuse it without modification, or remove and throw it away. The original system can be one or more reuse libraries and/or one or more software systems that are not in a library. When you add newly created code to the modified and reused code, you create a new software system. The relative proportions of these code types will vary from system to system. A "new" system can be either completely new code or a new version of an existent one, i.e., a mixture of new and reused code.

The code counting definitions and categories are (Gaffney and Cruickshank 1991):

new = added + modified
deleted = modified + removed
reused = original - deleted

These definitions do not imply that you must estimate any of the code types in any particular way. They are just general rules for categorizing counted code. Furthermore, these definitions do not imply what level of code you must count. A software metrics standard may require that you count several levels of code such as source statements and object statements. This guidebook strongly recommends counting source statements.

Rules for counting either logical source statements (*LSS*es) or physical source statements (*PSS*es) are listed in (Gaffney and Pietrolewicz 1990) and condensed here. This guidebook recommends that you

count LSSes, but you may build rules for both into your code counting facility. A source statement is anything that a programmer writes except comments. You should count all source statements and declarative, executable, and data definitions. Comments, if counted, should be counted separately.

You should always make separate code counts for each computer program unit with an identification of the function or program unit, the level, the code type, the type of count (LSS or PSS), and the language. You should make separate counts for source statements and comments. It is optional to count a blank comment line. You should not count noncomment blank lines used as separators.

The (cost) equivalent to new source lines of code, or *ESLOC* metric, combines all of the above types of source statements into one size metric. You can define the metric either in terms of SLOC or KSLOC (one thousand source lines of code). It combines new code (N) and reused code (R) sizes into one size metric that is cost-equivalent to all new code. The definition is:

$$\text{ESLOC} = (\text{SLOC})_N + w(\text{SLOC})_R$$

$$\text{KESLOC} = (\text{KSLOC})_N + w(\text{KSLOC})_R$$

Experience shows that w typically has a value of 0.04 to 0.31, which means that reused code costs (on a per statement or per unit basis) 0.04 to 0.31 as much as new code to be included in the system being developed.

Using a value of 0.30, a software product composed of 100 KSLOC of new code and 190 KSLOC of reused code has $100+(0.30)190=157$ *KESLOC* (thousand equivalent source lines of code). This example illustrates the representation of size by one number when there are two code types present.

The ESLOC and KSLOC concepts allow you to compute overall size and labor rate or productivity metrics (see Section 4.6 for definitions) for software products that include both new and reused code. The ESLOC concept facilitates size and productivity comparisons among computer programs and among development projects with varying code mixtures.

Your organization should determine a standard value for the weight w so that all definitions and calculations of ESLOC will be on the same basis and will be comparable. This parameter has a value of less than one, since reused code costs less than new code. The precise value of w will be a characteristic of the development environment and process. Section 4.3 discusses the computation of w as well as the derivation of labor rates.

4.3 DERIVATION OF LABOR RATES FOR NEW AND REUSED CODE

The cost of software development can be expressed as:

$$C = C_N + C_R$$

where C is the cost in LM (or LH) of the project, C_N is the cost of new code, and C_R is the cost of reused code. This cost includes all activities from design through integration and system test. Documentation costs (contract deliverables) are included, but support costs are not included. You can express C_N and C_R as the product of a labor rate in LM/KSLOC (or LH/SLOC) and a size in KSLOC (or SLOC) for new and reused code, respectively, as:

$$C = R_N K_N + R_R K_R$$

where R is the labor rate in LM/KSLOC and K is the size in KSLOC. In the case where new and reused code are both involved in the software product development, the separate rates are not observable for that project, although the overall cost of development and the sizes are observable. (The overall cost comes from accounting records, and the sizes result from counts of the SLOC.) Since the labor rates are not observable, they are unknowns. However, you can compute them as follows:

Denote the unknown labor rate parameters by u and v, and consider project j. The general equation is:

$$C_j = uK_{N,j} + vK_{R,j}$$

Assume that data on software size and cost for m development projects are available. Since you are to calculate two parameters, u and v, you need a minimum of data from two projects. However, if more than two project data sets are available, you should use all of the applicable data sets (i.e., those from similar development projects) since greater accuracy in the calculated labor rates will result. Let this data be formulated in matrix form as follows:

$$X = \begin{bmatrix} K_{N,1} & K_{R,1} \\ K_{N,2} & K_{R,2} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ K_{N,m} & K_{R,m} \end{bmatrix} \qquad Y = \begin{bmatrix} C_1 \\ C_2 \\ \cdot \\ \cdot \\ \cdot \\ C_m \end{bmatrix} \qquad B = \begin{bmatrix} u \\ v \end{bmatrix}$$

where $K_{N,j}$ and $K_{R,j}$ are the sizes in KSLOC of new and reused code in project j, and C_j is the cost of software development in project j. You can use $S_{N,j}$ and $S_{R,j}$ if you do the computation in SLOC.

Then you may solve the matrix equations (Graybill 1961) to get the values of u and v.

$$(X^TX)B = X^TY$$

$$B = (X^TX)^{-1}X^TY$$

Since the weight for reused code is relative to new code, the weight for reused code in the ESLOC relation is (v/u), and you then calculate ESLOC and KESLOC as:

$$ESLOC = S_N + (v/u)S_R$$

$$KESLOC = K_N + (v/u)K_R$$

The ratio v/u = w was defined in the previous section.

Remember that u and v are not just weights but are also labor rates that you can use for cost estimation. You can apply this mathematical technique, using all project cost and size data, to each phase of development (e.g., design, implementation, and test) or to each group of phases of development, and you can compute separate labor rates for each of these phases. Furthermore, you can apply the technique, using all project data, to the individual activities of development (if cost and size data are available) and compute separate labor rates for activities.

4.4 PRODUCT SIZE METRICS

Knowledge of the size of the planned software product is important since the size, however measured, will be one of the main determinants of resource allocation (dollars and personnel) to the development of the software product. The following list of data items are measures of software size because they are directly observable. They are also size metrics because you can use them, without any transformation, to represent software size.

You may measure the product size for the various software development activities by the following quantities:

- Requirements
  - Number of requirements
  - Number of (function) boxes in system diagrams
  - Number of (hardware) boxes in a computer network diagram
  - Number of major subjects or headings in a system description document
- Design
  - Number of design statements (source lines of design [*SLOD*]), program design language (*PDL*) statements, and structured narrative statements
  - Number of pages of design
  - Number of process "bubbles"
  - Number of boxes or arrows in hierarchical input-process-output (*HIPO*) charts
- Code
  - Number of source statements, i.e., SLOC by language, new, added, modified, reused, PSS, or LSS
  - Number of source statements, delivered and not delivered, by language
  - Comments (casual, headers, prologs)
  - Number of object code instructions or bytes
  - Number of words of memory
  - Number of screens
  - Number of operators and operands
  - Number of tokens
- Testing
  - Number of tests
  - Number of test procedure steps

- Functions
  - Number of CSCIs
  - Number of CSCs
  - Number of CSUs
  - Number of hardware boxes (where there is primary computer software controlling remote secondary computer hardware functions)
  - Number of inputs and outputs
  - Number of function points
  - Number of iterations per phase/activity combination in the Evolutionary Spiral Model
- Documentation: Number of pages of documentation (text, tables, figures) by document name or type

In most instances, you can find the number of requirements by counting the number of "shalls" in the specification. This guidebook assumes no formal mathematical notation for requirements specification.

You should count SLOD when you use a PDL or equivalent design method such as pseudo-code or structured narrative. Count SLOD for both preliminary design and detailed design. Derive SLOC/SLOD ratios for use in estimating software size in SLOC when you know only SLOD in design. Section 5 discusses this sizing technique.

You may use the list above to select the size metrics to use for tracking the size of the software through the development life cycle and to compute costs. You can find additional data items in (National Aeronautics and Space Administration 1990; IEEE 1990; Humphrey, Kitson, and Kasse 1989; and Grady and Caswell 1987) if you want to expand the list or if you want to make your own list of software size metrics. Section 4.9.2 recommends a minimum data set.

4.5 COST METRICS

4.5.1 Labor Cost Metrics

Labor is at least 80 percent of software development costs; therefore, cost metrics play a crucial role not only in the estimation of software development and maintenance costs but also in the management of software development. You should use the size metrics presented in Section 4.4 with the cost metrics to produce estimates of software development costs. The accurate estimation of software costs is widely acknowledged to be a critical problem in the management of software development, so cost metrics are of great importance.

You can measure software costs by LM, LH, and dollars. In estimating costs, you should estimate LM or LH first and then convert them to dollars when appropriate. LM and LH are the primary measures of cost since the use of these cost measures facilitates comparisons among different projects at different times. You should state the costs of activities that comprise the software development life cycle

in terms of LM or LH. You can state these cost measures in terms of dollars once you determine the proper labor categories.

You can compare measurements of cost in LM or LH with other measurements in LM or LH that you made at different points in time, such as with previous projects. Measurements made in dollars are difficult to compare through time or among development projects done by different organizations because of variability in labor rates (in dollars) among organizations and in the value of the dollar and because of the changing cost of labor. You should first make estimates in LM or LH and convert them to dollars for proposal and budgetary purposes. Normally, you use LM or LH as the primary estimation quantity.

It is important to note that where you use LM as the measure of effort, you should know and clearly state the number of hours per LM. Such a statement in cost reports facilitates the comparison and standardization of the LM measure among separate organizations.

A full LM is defined here as 167 hours, i.e., 2,000 hours of actual work (exclusive of holidays) per year divided by 12. A staffing LM is about 152 hours if you deduct the apportioned amounts of vacation, sick time, holidays, and (nonproject) lost time from the full LM. A full-time employee charges about 167 hours per LM to the project when working, but for staffing purposes, you must plan for the employee being at work about 152 (chargeable) hours per LM. Vacation, sick time, holidays, and lost time are nonchargeable to the project. Thus, use 167 hours per LM for calculating project labor costs and 152 hours per LM to plan the number of staff hours available to the project (the staffing level). The information presented in this guidebook is on the basis of 167 hours per LM, but if the definition of an LM is a problem, use LH.

The general rule in the definition and application of LM is that if you are computing costs, use 167 hours per LM. If you are computing schedules and staffing, use 152 hours per LM.

You should collect cost data in LM or LH for the total labor expended, normal time plus overtime. You should keep separate cost accounts for each CSCI in the WBS, and you should keep a separate cost account for each of the activities in the development process in the WBS. The development activities' cost accounts should tier up to the CSCI total development cost account, and there should be a separate tiering structure for each CSCI. If parts of a CSCI use different development processes, then each process should be a separate tiering structure. Such an accounting scheme makes budgeting more precise and makes the collection of costs much easier.

This guidebook presents costs in terms of LM and LH, not dollars, since dollars per LM or LH differ substantially among organizations, among locations, and by labor category. This guidebook presents unit costs in terms of LM/KSLOC or LH/SLOC. Unit costs and labor rates are synonymous.

4.5.2 Computer Usage Cost Metrics

The cost of computer support during software development is a measure that you should collect if your project is charged for computer time. Some software development projects use both host and target computer systems. In many cases, the target computer is government-furnished equipment (*GFE*), but the host computer is a corporate computer. No charge is made for the use of the GFE system, but an internal procedure exists that bills you for host computer time. Since the host system performs such tasks as simulation studies, data analysis, and program conversion for execution on the GFE system, this cost can be a significant burden.

Most billing procedures are based on the amount of computer time used, but this is a difficult quantity to estimate in advance of execution since few developers know how fast their software will execute on the host or target system and how many runs they will need to make. If your organization keeps accounting records on hours of computer time (internally or externally) billed and the LM or LH for software development, then you can derive an hours/LM or hours/LH metric. You can use this type of metric for estimating future development computer costs by using the estimate of LM or LH for development and multiplying by the appropriate ratio.

More commonly, organizations keep accounting records for computer usage (where internal billing is involved) in dollars. In this case, you should derive dollars/LM or dollars/LH metrics for computer usage. Furthermore, you can and should develop separate metrics of dollars/LM or dollars/LH for the coding, software integration testing, and system integration testing phases of development. The values of these metrics will differ, but the use of unique computer cost ratio metrics for each activity in software development can lead to very accurate estimates of computer costs. There should be a date associated with each computer cost metric to indicate the time when the cost metrics were updated. This time label will allow the adjustment of costs forward and backward in time.

4.6 PRODUCTIVITIES AND LABOR RATES

Use the size metrics from Section 4.4 with the cost metrics from Section 4.5 to produce productivities or their inverse, labor rates. Measure the productivities for the various activities in software development by:

- SLOC per LM or per LH (for either new or reused code).
- SLOD per LM or per LH.
- ESLOC per LM or per LH.
- KSLOC per LM.
- Words of object code per LM or per LH.
- Test procedure steps per LM or per LH.
- Pages of documentation per LM or per LH.
- Function points per LM or per LH.

You may use SLOC/LM as the productivity metric for the activities in development even though SLOC is not the output of some of the activities such as design. The SLOC to consider in these cases is the SLOC in the eventual size of the software, whether actual (counted) or estimated. Alternatively, the low-level design cost metric could be SLOD/LM. This metric requires estimates and counts of SLOD.

Labor rates are the scaled inverse of productivities. They are the inverse ratio of the list above, e.g., LH/SLOC. Alternatively, you can use LM/KSLOC as a labor rate metric for either new or reused code. The advantage of using LH/SLOC or LM/KSLOC is that these labor rates are additive so that you can easily calculate the total labor rate for development from the individual activity labor rates. Productivities such as SLOC/LM are not additive and thus offer no such convenience of representation

or calculation. Alternatively, you may use LM/KESLOC or LH/ESLOC as overall labor rate metrics. Labor rate metrics such as LM/KSLOC and LH/SLOC are also called unit costs.

The conversion formulas between productivities and labor rates are:

$$\text{SLOC/LM} = 1{,}000/(\text{LM/KSLOC})$$

$$\text{SLOC/LH} = 1/(\text{LH/SLOC})$$

Your software development organization should derive a labor rate for every activity in its development process. You can view this set of labor rates as the "standard" or "baseline" set to be used as guidance or as a "menu" corresponding to the set of possible activities composing the software development process. This process would consist of selecting the activities that form the standard or baseline set that describe (model) the particular software process in question and then modifying the standard labor rates, if necessary, for some of these selected activities.

If you use the spiral process (Boehm 1988), you should collect the number of iterations (or the number of prototypes) through the software development life cycle and the number of iterations through each of the activities that compose the spiral process. These quantities appear in the list of product size metrics given in Section 4.4. You should also collect the cost per iteration and the cost of the activities that compose each iteration.

When you collect labor cost data, whether in LM or LH, it is important that you collect all of the applicable labor cost data. Since a software product is often produced with significant overtime as well as normal working time, you should capture all of this cost data. To consider only normal working hours tends to overstate productivity and understate costs.

4.7 QUALITY MEASURES AND METRICS

Quality of the software product and of the software development process is an important consideration, and the basic quality measure is defects. To produce a high-quality product, you must attune your development process to a zero-defect method of operation. To produce a high-quality product at the lowest possible cost, you must remove defects at the earliest possible point in the development process. Thus, although testing with the related metrics are important, inspections and reviews and the metrics related to inspections and reviews are also important. Quality metrics include:

- Defects/KSLOC discovered in inspections (or reviews): preliminary design, detailed design, and code.
- Dollars/defect.
- LM or LH per defect.
- Number of each type of defect.
- Criticality of each defect (major or minor).
- Type of inspection (or review): preliminary design, detailed design, and code.
- Total labor hours of preparation.

- Number of persons participating in reviews or inspections.
- Total labor hours in inspection.
- Total statements inspected.
- Number of valid program trouble reports (*PTRs*) resulting from CSC integration test and CSCI test.
- Number of field error (deficiency) reports.

Possible categories for types of defects include design, logic, syntax, standards, data, return message, prolog/comment, performance, and interface. You may count requirements defects, both the number of instances of incompleteness and the number of instances of inconsistencies, if requirements reviews are a formal part of your software development process. Otherwise, the systems engineering organization should handle requirements defects. This guidebook assumes no formal mathematical notation for requirements specification.

4.8 SCHEDULE MEASURES

Schedule is an important determinant of cost. If not enough time is available for the development of a software product, you will make errors and not discover them, and your costs will be high. If too much time is available, your productivity will be low and your costs will be high. It is important to have a reasonable schedule for development.

The most common schedule measurable is time in months, although you might use years. You may also use milestones and events as time or schedule measures.

4.9 THE ENTERPRISE SOFTWARE EXPERIENCE DATABASE

4.9.1 The Enterprise

"Enterprise" is a term that refers to the organization that sponsors one or more software development organizations. A large company composed of large divisions should consider each of its divisions to be separate enterprises, since software standards are usually controlled at the division level. Alternatively, a plant and laboratory location could be an enterprise, since such sites are usually (like the division) a profit center. Each such enterprise should establish and maintain a software experience database as a permanent repository for all of its software experience, i.e., the actual product costs, sizes, quality, and process activity and schedule measurements that characterize the product and process at the time of project completion.

The establishment and maintenance of an enterprise software experience database should be an activity conducted at the highest levels of the enterprise, since many projects and organizations will contribute information to it. Software development, systems engineering, system test, quality engineering, configuration management, product support (logistics), and project management are among the organizations that will contribute to the database and are among the organizations that will benefit from the software information it contains.

You can use the software experience database to "feed back" the more precise data about your enterprise's past performance, help improve the software and management processes, better define costs, and improve the planning and proposal processes. An enterprise software experience database is a powerful tool for improving organizational performance, and every enterprise should establish such a database.

4.9.2 Minimum Data Set

The minimum data set is the smallest number of data items required to adequately represent project software development experience. The minimum data set represents the smallest number of data items that, when collected, analyzed, and fed back to the process, will produce improvement in the development process. You should collect this data set and enter it into the enterprise software experience database.

You should collect the actual values of the following software size, cost, quality, and schedule measurements upon project completion and enter them into an enterprise software experience database as a minimum data set. You can calculate project metrics such as labor rates (or productivities) and defects/KSLOC from this set of measurements. You can use these measurements and metrics not only to evaluate the software development performance of the project from which they came, but also to establish cost estimation standards and project control methods for future projects.

- Size:
  - Number of CSUs, CSCs, and CSCIs.
  - Number of SLOD by CSC and CSCI.
  - Number of SLOC by CSC and CSCI, both initial estimate and final count.
  - Number of words of memory used by the executable code developed.
  - Languages used.
  - Number of prototype and iterations in spiral model development (where applicable).
  - Number of tests and/or test procedure steps for CSC and CSCI tests.
  - Number of function points (where applicable).
  - Number of pages of documentation (by document).
- Cost: The cost in dollars and in LM or LH for each CSC and CSCI by development activity and by iteration. (You should collect labor cost data for both normal and overtime hours. You may collect dollar cost separately for labor, development computer time, subcontractors, and supplies if desired.)
- Quality: From each type of inspection, preliminary design (internal) and PDRs, detailed design (internal) and CDRs, and code inspections, collect the following measurements:
  - Number of defects found per review or inspection.
  - Number of SLOD reviewed.

  - Number of SLOC inspected.
  - Number of valid PTRs by test activity.
  - Number of PTRs closed.

- Schedule: Actual initiation and completion dates by activity for each CSC and CSCI.
- Model and type of both host and target computer systems.

You should count the number of defects found in requirements reviews only if requirements reviews are a formal part of your software development process. Otherwise, the systems engineering organization should handle requirements reviews and defects. In most cases, you can find the number of requirements by counting the number of "shalls" in the specification.

You should count the SLOD when you use a PDL or equivalent design method such as pseudo-code or structured narrative. Count SLOD for both preliminary design and detailed design. Derive SLOC/SLOD ratios for use in estimating software size in SLOC when you know only SLOD in design.

You should collect and store at least the minimum data set in the enterprise software experience database. Your enterprise should define and develop its own data set appropriate for deriving metrics values tuned to its software development environment.

4.9.3 Measurements Collection

The metrics group should collect the data, and this activity involves a hands-on approach. The group with data collection responsibility should not send out forms and expect any sort of reliable data to come back. If the forms are filled out at all, the data will be incomplete, inaccurate, and biased toward making the project performance look good. No matter how many pronouncements management sends out stating the importance of proper data reporting, it will happen with great difficulty or not at all. Project personnel will not voluntarily provide "actuals" that are actual. The metrics group must collect the data by personally reviewing project and accounting records.

You should collect the metrics data in real time, i.e., at the planned time during the project and immediately upon completion of the project. It is a mistake to try to reconstruct measurement after the fact, i.e., when the project records are gone. Also, it will be very difficult to collect the proper metrics data at the end of a project if the metrics group has not been tracking the project all the way through. The tracking and monitoring effort provides the metrics group with a basic familiarity of the processes, products, and problems of the development project, and thus enables efficient and precise data collection.

The group with the data collection responsibility should take pains to reassure project personnel, both technical and management, that they are collecting the data to evaluate it. They should repeatedly remind project personnel that the data will not affect their performance appraisals. To reinforce this statement, do not label the data with the names of any project personnel.

Most software development environments include a code counting ability, and you should use this ability to count source statements. You should count both logical and physical lines. Count all source statements, not just executables, and count comments separately from the source statements.

Table 4-1 (included at the end of this section) shows a data collection form that you can use to record measurements data upon completion of the project. The measurement data you enter on this form are the "actuals" for the project. Intermediate data collected while the project is ongoing (i.e., management indicator data) uses a different form, as shown in Section 9. You should collect measurements data for each CSCI and for each CSC if possible. The documents listed in the sample data collection form are those specified in DOD-STD-2167A. If you use another standard, then the documents will be different from the form.

4.10 SUMMARY OF RECOMMENDATIONS

The recommendations on data collection presented in Section 4 are:

- Collect LM or LH as well as dollars for cost performance. LM and LH should include overtime, even if unpaid.
- Count source statements. Count comments separately. Do not count noncomment blank lines.
- Establish an enterprise software experience database to aid in deriving software metrics and estimating. This database should contain the "actuals" (actual costs, counted sizes, etc.) from the process and product aspects of the project collected upon completion of the project.
- Begin data collection as soon as you institute a measurements program, regardless of process maturity level.
- Define the measurement data to be collected. Collect at least the minimum data set (or the equivalent as defined by your enterprise).
- Establish separate WBS cost accounts for each CSCI. Establish separate CSCI-specific cost accounts for each development activity or phase that tier up to each total CSCI cost account.
- Compute cost estimates and initially state them in LM or LH. The LM or LH mode allows more precise comparisons between projects or activities. The value of the dollar constantly changes through time and by labor category whereas LM or LH do not. Convert LM and LH to dollars when appropriate.

Table 4-1. Software Experience Database Measurements Collection Form

Project			CSCI				Date	
Language				Cost or Cost per Iteration			Month/Year	
CSC	Phase/Activity Indicate Iteration		Cost Account	LM	Computer $	Subcontract $	Start	End

CSC	Final Count: Indicate New (N) or Reused (R)				Defects From Each Inspection Level			PTRs	
	SLOD	SLOC (or Fn.Pts.)	Initial Estimate	CSUs	Prelim. Design	Detailed Design	Code	CSC Test	CSCI Test

Host Computer		Target Computer	

Table 4-1, continued

Document	Development data			
	Pages	LM/LH	Start Date	End Date
System/segment specification				
System/segment design document				
Software requirements specification				
Interface requirements specification				
Software development plan				
Software test plan				
Software design document (preliminary)				
Software design document (detailed)				
Interface design document				
Software test description (cases)				
Software test description (procedures)				
Software test reports				
Version description document				
Software product specification				
Operation and support document				

This page intentionally left blank.

5. HOW TO ESTIMATE SOFTWARE SYSTEM SIZE

5.1 SIZE ESTIMATION

5.1.1 Size Estimation Activities

This section answers the question, How big will the system be? It addresses the need for methods to estimate software size, especially methods that you can apply early in the development cycle and at each milestone in the development life cycle.

Estimating the size of a new software system is key to estimating development cost, both in dollar terms and the amount of labor and other resources required, since the size determines a large part of the cost of a software system. An accurate estimate of size is important because size is the basis for estimates of development costs for software. Errors in the estimate of a software system's size often exceed the estimation error for the amount of labor required for development and for productivity of its creation.

Table 5-1 summarizes the size measurement activities and indicates which activities are appropriate for the process maturity levels. It shows activities appropriate for use by organizations at process maturity levels 1 and 2 and activities suited to organizations at process maturity levels 3 through 5, based on the fact that more experience data is available to organizations at levels 3 and above.

Table 5-1. Measurement-Related Activities by Process Maturity Level

Levels 1 and 2: Repeatable Process	**Levels 3 Through 5: Level 2 Data Plus:**
Estimate, plan, and measure software size, resource estimates, staffing levels, schedules, and development cost. Maintain profiles over time for units designed, units completing test, units integrated, requirements changes, and staffing levels. Collect statistics on design errors and on software code and test errors found in reviews and inspections. Maintain profiles over time for utilization of target system memory, throughput, and I/O channels.	Maintain a managed and controlled process database for process metrics across all projects. Maintain formal records for the progress of unit development. Routinely project, compare to actuals, and analyze errors found in reviews and inspections of requirements, designs, and code and test.

You should reestimate code size throughout the development process. Derive the initial estimate from the requirements. You can derive subsequent estimates during the development process based on increased knowledge about the software system as it evolves. Derive this information from further elaboration and expansion of the requirements, and on design and other intermediate products of the

software development process, as you create them during development. The methodology presented here uses knowledge about what is to be built (the requirements) as well as what has been built so far during the development process. However, the emphasis is on the development of "front-end" estimates, those you make before you begin the development process.

There are two principal types of measures of software product size:

- ***The Amount of Code.*** This is the number of source statements or number of SLOC, as discussed in Section 4. An SLOC count tends to vary with the language in which the software is written. That is, in general, a given system is expressible in fewer higher level language statements (e.g., Ada) than are required for lower level (e.g., assembly) language.

- ***The Number of Function Points.*** This is a measure of the amount of function provided by the software system, as defined in Section 5.5. It is independent of the language in which the system is written.This section focuses on the estimation of code size in SLOC because it is the more general approach.

5.1.2 Size Estimation and the Development Life Cycle

This section is concerned with estimating the size of a new software system very early in the development life cycle, i.e., in the conceptual and requirements phases. At this very early point in the development cycle, there is little detailed information available about the software system. In fact, the planned functions have not usually been assigned to new and reused code. Later in the development cycle, when functions have been assigned to new and reused software and when more information is available about the software, you can estimate the size of the functions to be represented by new code using other methods presented in this section. When you identify the functions to be represented by reused code, you can count the reused code.

5.1.3 Size Estimation and Process Maturity Levels

Table 5-2 summarizes relevant size estimation methods appropriate for organizations at process maturity levels 1 and 2 and for organizations at process maturity levels 3 through 5. The level 1 and 2 processes concentrate on using methods that require a minimum of experience data. The level 3 through 5 methods emphasize the use of experience data to estimate size, incorporating lessons learned from results of the organization's process modifications and development experience.

Software size is the primary parametric input to most cost and schedule estimating algorithms, but the estimates of size used for these purposes are frequently inaccurate. Inaccuracies in size estimates are the primary cause for inaccuracies in cost and schedule estimates. Even where you know unit costs and where you have thoroughly analyzed schedule constraints, the overall cost and schedule estimates for many software development projects may be inaccurate because you did not base your size estimates on quantitative measurement. You need systematic methods for size estimation.

Code size estimates are based in part on past experience. Use your own organization's experience. This experience is accessible in terms of the data available in a formal database. For a large number of systems, it contains the sizes of CSCIs, the number of major inputs and outputs, the code counts, and other such information. Take considerable care in making a direct analogy between projects, a new one and another for which information is available in a database.

Table 5-2. Methods for Estimating Software Size

Life Cycle Phase	SEI Process Maturity Level 1 or 2	SEI Process Maturity Levels 3 Through 5
Conceptual	Count function blocks and convert to SLOC; compare system with known similar systems; convert CSCIs to SLOC.	Count function blocks and convert to SLOC; make detailed comparison of system with known similar systems.
Proposal	Count I/O (externals); compare with known similar systems; make statistical size estimation; convert CSCIs to SLOC.	Count I/O (externals); make detailed comparison with known similar systems; make statistical size estimation.
Requirements	Count externals; compare with known similar systems; make statistical size estimation; convert CSCIs to SLOC.	Count externals; make detailed comparison with known similar systems; make statistical size estimation.
Design	Convert the number of CSCIs and the number of CSCs to SLOC.	Convert SLOD to SLOC; convert number of CSCIs and CSCs to SLOC.
Coding	Determine code counts and establish configuration management library.	Determine code counts and establish a configuration management library and development environment.
Testing	Determine code counts and establish configuration management library.	Determine code counts and establish configuration management library.
Overall	Be aware of tendency toward code growth during development process.	Estimate code growth using factors from organization's process database.

5.2 SIZE ESTIMATION DURING THE DEVELOPMENT LIFE CYCLE

5.2.1 Size Estimation by Development Phase

An enterprise with a defined software development process (i.e., the process of an organization at SEI process maturity level 3) should have accumulated enough data in its software experience database that it can make reasonable estimates of size throughout the life cycle of the software. The life cycle is composed of many activities, all of which you should define and place under management control at process maturity level 3. It is not necessary to make estimates of size as the product transitions from one development activity to another, but you should make size estimates at every phase during development. Here a phase is defined as a group of activities. For example, the design phase might consist of the activities of preliminary design, detailed design, internal design reviews (inspections), PDRs, and CDRs.

Monitor a software development project, with all of its associated software products, throughout the development life cycle using a continuous measurement process. Make measurements such as estimates of product size in the conceptual phase, at the requirements phase, at proposal time, at initiation of the project, and through the design coding and testing phases. The software product size estimates you make at the conceptual, proposal, and requirements phases are characterized by considerable variation because so little detail about the software is available. Size estimates you make at the design, code, and test phases are subject to much less variation since they are based primarily on code counts ("actuals"). The process of measurement and size estimation through the development life cycle is characterized by the increasing accuracy of size estimates.

In general, when your software development project is at the system-level conceptual phase, you should use a technique such as function block counting (see Section 5.3) since you know very little

else but major functions included in the system at this stage. As the project moves from the conceptual phase to the proposal phase, techniques like I/O counting (see Section 5.6) come into play since you will have defined the main interfaces between the major functions. At the proposal phase, you can use function block and I/O techniques as cross-checks on each other to achieve a reasonable level of confidence that you made a suitable size estimate. At the design phase, you should count design statements and convert them to SLOC estimates by an SLOC-to-SLOD ratio derived from your enterprise's experience. As the project moves from design to coding and testing, use code counts to give very accurate estimates of cost. Finally, at the testing phase, code growth becomes important. Section 5 presents size estimation methods applicable at all of these points in the development life cycle.

5.2.2 Using Source Lines of Design to Estimate Software Size

In the design phase, information in the form of counts of SLOD are available if you used a design language. Preserve these counts of SLOD in the enterprise software experience database. When the corresponding SLOC counts become available, compute an SLOC-to-SLOD ratio and preserve it. This ratio will aid you in converting SLOD to SLOC estimates for future projects. Alternatively, you can compute function point-to-SLOD ratios and use them for future estimates of the amount of design (in SLOD) that will be developed for a system of some given size in function points.

5.2.3 Size Estimation Steps

The following steps apply to size estimation made at any stage in the development life cycle:

- At the conceptual or requirements phases, compile the requirements for the system (or part of a system, such as CSCI) whose size you wish to estimate.
- Get all of the data you can about the system you are to estimate (i.e., functions to be performed, counts of inputs and outputs, etc.).
- Get as much detail as you can. Get the data for as many of the (likely) component functions or parts of the system as you can. Consult with those who have the most expert knowledge of the components or functions in which you are interested. Take advantage of the fact that an underestimate of the size of one component often cancels the overestimate of the size of another component.
- Perform several estimates at a time based on different types of data. Compare them. If they differ beyond, say, 20 percent, reconsider your assumptions and try again. Repeat the process.

5.3 FUNCTION BLOCK COUNTING

This section describes a method for providing a rough estimate of software system size when you have very little information about the intended application system. All that is required is the (estimated) number of CSCIs or CSCs. This method has the advantage of requiring little information, so you can apply it very early in the development cycle. You can apply this method at the conceptualization or requirements stages when the size implication of a requirement is itself the input to a decision-making process, e.g., the decision to include the function or to bid on a contract. The method has the disadvantage of not directly including the effect of other information that may be available about the application system.

You can count major functions as the number of CSCIs or the number of CSCs. You can also count major functions as function blocks in a system description document or in a system-level block diagram or flow chart. At the conceptual phase (when the system is being planned at the top level), you should identify the major software functions and have them appear as function blocks. A function block corresponds to a CSCI, and the next level of decomposition, major subfunctions, corresponds to a CSC. This section describes a method for estimating the size of a software system based on the number of major software functions or subfunctions that are expected to compose the system.

You can apply the methods presented here at one or two levels of decomposition, at the CSCI or CSC level. (Gaffney 1984) and (Britcher and Gaffney 1985) demonstrated that, under the assumption that most systems have the same number of levels of decomposition, you can estimate the size of the system as a function of the number of functional elements at that level. Thus you would not expect a larger system to have more (vertical) levels of decomposition than a smaller one. Rather, it would have more units of code at each level, CSCI and CSC.

You can base your estimate of software system size on the number of CSCs or CSCIs that you expect to compose the system. (Britcher and Gaffney 1985) present figures of 41.6 KSLOC and 4.16 KSLOC, respectively, for the expected values of size for a CSCI and a CSC. Experience suggests that the size of a CSCI can vary rather substantially among systems or even within a given system. It is quite reasonable to expect a substantial variation since the estimation process uses little information about the actual system. Some data on the experience of an aerospace contractor showed that the standard deviation of the sizes of a CSCI averaged 27.5 percent of the expected value, $\sigma/E = 0.275$, using the notation of Section 5.4. The values presented here are undoubtedly domain-dependent, and you should make great efforts to adopt these values to the domain or environment in question.

Your enterprise should collect data about the sizes of the CSCI and CSC and then develop average size figures to use in developing function block estimates as described here. However, if no such data is available, then you might use the figures of 41.6 KSLOC and 11.45 KSLOC ($= 0.275 \times 41.6$), respectively, for the expected value and standard deviation of the size of each CSCI in the system whose size you wish to estimate. Then, based on the statistical concepts presented in Section 5.4, calculate the expected estimate of size in KSLOC and the estimated standard deviation of the size in KSLOC of the overall system using the equations:

$$E_{tot} = 41.6 * N$$

$$\sigma_{tot} = 0.275 * E_{tot}$$

where N is the number of CSCIs. Alternatively, you can multiply the number of CSCs by 4.16 KSLOC to produce an estimate of size. You can use the two size estimates as a cross-check on each other.

In summary, the steps of the function block method are:

- Count or estimate the number of blocks at a given level of detail, i.e., at the CSCI or CSC level, or both.
- Multiply the number of blocks times the expected value of the size for that type of block. This is the expected size of the system overall.
- Compute the standard deviation of estimated system size.

- Compute the desired range of the system size for the probability levels desired per the method described in Section 5.4.
- Apply this method for both the count of CSCIs and for the count of CSCs and pool the results. Do not apply this method when there are fewer than three function blocks.

5.4 STATISTICAL SIZE ESTIMATION

This section describes a systematic method for estimating the code size of a software system by estimating the ranges of size of the component elements such as CSCs and CSCIs that will compose it. It is a method for systematically making estimates of the sizes of the software system's individual components that are to be developed and then accumulating them. Decomposing the system into a number of functions, considering each of them in turn, and then statistically operating on the data to obtain estimates of the overall size and its standard deviation enables you to reduce the effect of uncertainty in the size of the individual components and to obtain a better estimate of the size of the system overall. The source of the information for the size estimate would typically be the sizes of components or units in similar jobs that your organization has done earlier, i.e., sizes of software products from the enterprise software experience database.

The method presented here systematizes the estimation-by-analogy approach based on enterprise experience. Such estimates are done by an individual or by pooling the educated guesses of a group of people. The method is described in (Putnam 1978). The steps in this process are:

- Determine the functions that will compose the new system.
- Compile size data about any similar functions previously developed.
- Identify the differences between the similar functions and the new ones.
- For each component function whose size is to be estimated, estimate three parameters:
  - The lowest possible number of source statements (or function points or other measure).
  - The highest possible number of source statements (or function points or other measure).
  - The most likely number of source statements (or function points or other size measure).
- Compute two numbers for the estimated size of each of the components, the expected value and the standard deviation. The formulas for calculating each of them are as follows:
  - The equation for estimating the expected value of the number of source statements (or function points or other size measure) in the *i*th unit of code, E_i, is:

$$E_i = \frac{a_i + 4m_i + b_i}{6}$$

    where a_i is the lowest possible number, b_i is the highest possible number, and m_i is the most likely number.

  - The equation for estimating the standard deviation of the number of source statements (or function points or other size measure) in the *i*th unit of code, σ_i, is:

$$\sigma_i = \frac{|b_i - a_i|}{6}$$

- Tabulate the estimates for each of the components.
- Compute the expected value, E_{tot}, and the standard deviation, σ_{tot}, for the overall system.

$$E_{tot} = \sum_{i=1}^{n} E_i$$

$$\sigma_{tot} = \left(\sum_{i=1}^{n} \sigma_i^2 \right)^{1/2}$$

Table 5-3 is an example of a table that you can use when applying the method just described. Illustrated is a case in which there are four units of software to be built.

Table 5-3. Size Estimation Table Example

Function	Smallest	Most Likely	Largest	Expected	Standard Deviation
A	5,830	8,962	17,750	9,905	1,987
B	9,375	14,625	28,000	15,979	3,104
C	6,300	13,700	36,250	16,225	4,992
D	5,875	8,975	14,625	9,400	1,458
Overall				51,509	6,374

You can approximate the uncertainty in the overall size of the system using the values just calculated for the overall expected value and standard deviation under the assumption that the size is normally distributed. You would expect this approximation to be more accurate for cases in which there are larger numbers of functions in the overall system. Some of the size uncertainty ranges are:

68 percent range: $E_{tot} \pm 1\,\sigma$: 45,135 to 57,883

99 percent range: $E_{tot} \pm 3\,\sigma$: 32,387 to 70,631

The 99 percent probability range is much wider than the 68 percent range.

You can use the method described here to provide a range of size estimates for use in the calculation of cost risk as described in the Section 5.5.

5.5 FUNCTION POINTS

The function point measure intends to measure the functionality of the software product in standard units, independent of the coding language. A function point is a measure derived from the (estimated) number of "externals" (inputs, outputs, control messages, inquiries, and interfaces) of a system and from the estimated number of internal files of a program unit. This section briefly describes the nature of the function point measure and summarizes how you calculate it.

You begin calculating the function point measure of a system by counting the number of each of the four types of system externals, plus the count of internal logical files. The statement of software system

requirements often describes the externally visible behavior of the intended system. The four types of externals are just such items:

- External inputs: Unique data and/or control inputs that enter the external boundary of the system which cause processing to take place. Specific examples include input files (data files, control files), input tables, input forms (documents, data entry sheets), input screens (data screens, functional screens), and input transactions (control discretes, interrupts, system messages, and error messages).

- External outputs: Unique data and/or control outputs that leave the external boundary of the system after processing has occurred. Specific examples include output files (data files, control files), output tables, output reports (printed and screen reports from a single interrupt, system messages, and error messages).

- External inquiries: Unique I/O queries which require an immediate response. Specific examples include prompts, interrupts, and calls.

- External interfaces: Unique files or programs which are passed across the external boundary of the system. Specific examples include common utilities (I/O routines, sorting algorithms), math libraries (library of matrix manipulation routines, library of coordinate conversion routines), program libraries (run-time libraries, package or generic libraries), shared databases, and shared files.

- Internal files: A logical grouping of data or control information stored internal to the system. Specific examples include databases, logical files, control files, and directories.

Estimate the complexity of each type of external and of the internal logical file items: low, medium, or high. Then multiply each count by the appropriate weight shown in Table 5-4 and sum it to determine the "function count."

Table 5-4. Function Count Weights for Complexity

Description	Complexity Weights		
	Low	Medium	High
Input	3	4	6
Output	4	5	7
Inquiries	3	4	6
Interfaces	5	7	10
Files	7	10	15

The next step in the calculation of function points is to determine the "value adjustment factor" by assessing the impact of 14 factors relating to the operation of the system (likely, or actual in the case of an existent system). Finally, you calculate the function point count by multiplying the function count by the value adjustment factor.

More information about function points, including rules for calculating them, is given in (Albrecht 1979; Albrecht and Gaffney 1983; Brown 1990; and Jones 1990).

Function points correlate well with software cost, as do lines of code for management information systems *(MIS)* software. However, they have not been widely used for embedded projects. (Jones 1990) developed a variant on function points, called feature points, in an attempt to recognize the greater likelihood of complex algorithmic computations in embedded programs (which is not recognized by the procedure shown above). Jones has asserted that this modification of the function point measure makes it suitable for application to the estimation of real-time systems.

The function point size metric is consistent across languages and applications. You can use function points to estimate source code size in SLOC, when you know the SLOC-to-function point ratio, by multiplying that ratio by the number of function points. The enterprise software experience database should contain this information.

Even if you can calculate it relatively consistently and accurately, the function point number does not answer all of the questions you would want to know about size. It is likely that you can estimate the physical size of a program or major element thereof, such as a CSCI, relatively easily from a KSLOC estimate. To do this, use data about the compiler's functioning, in particular the average expansion from SLOC to the number of object statements. Then multiply this figure by the average size (in bytes) based on experience captured in the enterprise database, of an object statement. Unfortunately, this process is not likely to be done when function points are the measure of program size. This is because there is, in general, no fundamental (physical) program size to perform a given function. Indeed, (Boehm 1983) reported on an experiment in which there was a several-fold variation in the physical size of the programs designed to meet the same functional objectives but with different optimizing criteria.

Experience with MIS and commercial software using function points shows that an early estimate of size can be made, generally quite successfully, for those classes of software. However, function point advocates typically use the counts of the five items cited above as the basis for calculating function points, as described above, and not as the basis for making an estimate of the count of source statements.

5.6 HOW TO ESTIMATE SOFTWARE SIZE BY COUNTING EXTERNALS

This section shows how to estimate the size in KSLOC of an aerospace software system using external measures of the intended system's requirements. (Here aerospace software can be characterized as real-time command and control embedded software.) This method is of particular interest since the measures you use are counts that are often available very early in the development cycle. The method is a generalization of the function point method described in Section 5.5, and that section defines the four external measures.

The method described here was developed at the Consortium (Gaffney and Werling 1991) based on a sample of 19 aerospace software samples. This work suggests that the unweighted sum of the counts of the externals (the primitives from which the function point value is determined) correlates about as well with the source statement count as do function points. Since the calculation of function points involves a subjective estimation of some additional factors, including the appropriate weighting to apply to the counts of each of the primitives, use of the "raw" sum of the primitives could prove advantageous since it does not require making the additional subjective judgments implicit in the estimation of function points. Not doing the weighting and other processing of the raw counts is simpler and might result in a reduced degree of error in the source statement estimate determined from it.

You can apply an empirical software size estimating model, based on counts of the program externals defined here, to both embedded and business software systems. The estimates of the parameters of such a model are best developed by the organization intending to use it, based on data from the experience of that organization. However, if such data is not available, you can use either of the two estimating equations presented here, one for the case of three externals and the other for four externals.

Two estimating equations (Gaffney and Werling 1991) for estimating size from counts of externals are:

$$S = 13.94 + 0.034A$$

where S is the software system size in KSLOC and A is the sum of three program externals—inputs, outputs, and inquiries—and:

$$S = 12.28 + 0.030E$$

where S is the software system size in KSLOC and E is the sum of all four program externals plus the number of interfaces.

The estimation procedure is:

- Collect counts of program externals and product size in KSLOC for projects in your enterprise software experience database.
- Develop an organization-specific estimating formula for estimating size from counts of externals. To derive such a formula, use the project data in the enterprise software experience database to plot size (on the y-axis) against counts of externals (on the x-axis), and fit a line that seems to best represent the data. The fit can be visual or it can be a linear regression fit as described in (Graybill 1961) and other standard texts.
- To estimate size for a new proposal, identify the number of program externals (including control commands in external outputs) for each major program unit (CSCI or equivalent).
- Estimate size in KSLOC by using the formulas above for the appropriate counts of externals.

You can obtain the data to be used in the estimating model and use it to develop an estimate of software size for your project at requirements time. This allows you to make a more accurate estimate of development costs earlier in the project. You can pool this estimate with size estimates developed using other techniques.

5.7 SOFTWARE PRODUCT SIZE GROWTH

The size, however measured, of software products of all types tends to grow from the time you initiate development to the time you deliver the product. No matter how accurate the data used to make the initial estimate and no matter how precise the method used to make the initial method, the delivered size will differ significantly in most cases from the initial estimate of size. This growth adds to the development cost and thus becomes an important consideration.

The percent code growth can be defined as:

$$\text{Growth} = \frac{(\text{Delivered\_Size}) - (\text{Initial\_Estimate})}{(\text{Initial\_Estimate})} \cdot 100$$

where size can be measured in SLOC, function points, etc.

Code growth occurs because you almost always tend to underestimate size at the conceptual, proposal, and requirements phases of the software project. You tend to be optimistic and may not know or fully understand the requirements, and both of these factors cause underestimation. Since this code growth adds to cost, staffing, and schedule problems, you must measure it to understand it.

(Cruickshank 1985) gives some experience with code growth in aerospace software development. Table 5-5 summarizes this experience, based on 16 projects in the 200 to 400 KSLOC range.

Table 5-5. Code Growth Factors

Development Time in Months (Design to CSCI Test)	Percent Growth Initiation to Delivery
12	11
24	19
36	32
48	55

For example, you would expect a software development effort scheduled to take two years or 24 months from preliminary design through CSCI (functional) to grow 19 percent from the original estimate. If the predesign estimate is 320 SLOC, then the estimate of the delivered size will be (320)(1.19) = 380.8 KSLOC.

You should make estimates of delivered size and code growth, and you should make plans to deal with the predicted growth. You should also establish a reserve account to fund the costs resulting from code growth.

5.8 SUMMARY OF RECOMMENDATIONS

Table 5-6 summarizes the recommendations on size estimation that depend on process maturity levels presented in Section 5.

Table 5-6. Recommendations on Methods for Estimating Software Size

SEI Process Maturity Levels 1 and 2	SEI Process Maturity Levels 3 Through 5
Organize the enterprise software experience database. Collect data including size data.	Use experience data to make more accurate estimates of size as early as possible in the development life cycle.
Make estimates of size at each development phase.	Make estimates of size at each development phase using experience data.
Estimate size by statistical methods and by counts of externals.	Estimate size by converting counts of SLOD to SLOC and by using counts of externals.

The general recommendations on size estimation presented in Section 5 are:

- Count function blocks, CSCIs, and CSCs, and measure their size in SLOC or function points. Record this information in the enterprise software experience database and use it for future size estimates.
- Use function point measurements, where applicable, to estimate MIS software size.

- Use counts of externals, where applicable, to estimate software system size.
- Make estimates of code growth during development. Establish a reserve to fund costs resulting from code growth.
- Develop enterprise-specific and domain-specific parameter values and metrics values for use in size estimation.
- Develop an enterprise software experience database to aid in size estimation.

6. HOW TO ESTIMATE SOFTWARE COST

6.1 OVERVIEW

6.1.1 Cost Estimation Methods

This section shows you how to estimate software development costs using holistic models and activity-based models. Holistic models are overview models that yield overall estimates of software development labor cost. Activity-based models use a bottom-up approach to software development cost estimation based on an analysis of the costs of the individual activities that compose the software development process. This methodology is especially useful in an environment in which you have established an enterprise software experience database, and where you use that database to feed back information about the process to improve the process. This section presents a set of procedures that you can use with a variety of software development processes. It also presents cost risk estimation procedures which offer decision-making criteria useful throughout the life cycle.

A considerable amount of material in this section is devoted to the activity-based models. This approach is due to the fact that you are likely to have more familiarity with holistic models such as COCOMO (Constructive Cost Model) than with activity-based models. This section also presents additional procedures to estimate maintenance, documentation, and testing costs. It discusses risk estimation and management methods using examples.

The estimating methods presented in this section are guidelines. You can and should modify them as your judgment and experience indicates to be appropriate in specific cost estimating situations. As your process maturity level increases, the enterprise software experience database will provide you the information you need to use your own parameter values in the methods given. In addition, you should apply the estimating methods presented in this section in an iterative way, continuously throughout the development life cycle. You should apply each of these methods within the scope of the enterprise cost management policies, plans, and procedures.

Both the methods and the parameter values presented here are offered as guidance; you should not view them in an absolute sense. You should determine whether a method will prove useful in your environment and, if so, adopt the method to your particular estimating environment. Your adaptation can take the form of modifying the method and also of changing the parametric values to suit your particular environment.

6.1.2 Cost Estimation and Process Maturity Levels

The methods presented here represent a rough progression through the SEI process maturity levels. If your software development organization is at process maturity level 1, the initial level, you should

estimate costs with the holistic models shown in Section 6.2. By the time your software development organization is at SEI process maturity level 2, the repeatable process level, you will have established an enterprise software experience database and accumulated sufficient data so that you can calculate labor rates for the main activities of software development (e.g., requirements definition, design, code and test, integration and test) and apply them on a project basis, as shown in Section 6.3. When your organization reaches SEI process maturity level 3, the defined level, you will have sufficient experience and data to do accurate cost estimation, as shown in Section 6.3, as well to estimate documentation and support to software costs, as shown in Sections 6.4 and 6.8. When your organization reaches process maturity level 4, the managed level, you will have sufficient expertise to estimate testing costs, as shown in Section 6.5, and to use top-down methods, as shown in Section 6.7. Finally, at level 5, the optimized level, your organization will be sophisticated enough to routinely handle cost risk, as shown in Section 6.9, and the cost of software maintenance, as shown in Section 6.11.

In general, to reach process maturity level 2, you must have formal cost estimating procedures in place. These procedures should be the holistic models at level 1 and early level 2 and the activity-based models at late level 2 and level three. The quality data you collect at levels 1 and 2 will consist of project code and test error data including code inspection data and test defect reports. At level 3, you must have formal design review procedures in place so that you will collect project design errors from preliminary and detailed design reviews. You use this design defect data to improve the process at levels 4 and 5.

Level 4 is the process maturity level where you use data from the project level to improve the process. At this point, your thinking about software development switches from thinking about the products of individual projects to the organizational software process. In other words, you will put a standard process into place at level 4 to be optimized by continuous improvement at process maturity level 5. This process of continuous improvement from level 1 to level 5 is also reflected in the way that you collect the data and build the enterprise software experience database. At levels 1 and 2, the data will be mostly product data such as cost and size. At levels 3 and 4, you collect process data such as activity data and quality data for defects in addition to the cost and size data.

6.1.3 Units of Cost

The computation and initial statement of software costs presented in this section are in LM and LH. Use these units of cost to facilitate your comparison of costs among projects and among activities. Do not compute costs in dollars; the dollar changes value both through time and through the many varieties of labor rates. When you have generated and analyzed your software development labor costs, you can easily convert them to dollars.

6.2 HOLISTIC MODELS

A holistic model, applicable as early as levels 1 and 2, estimates software (development) cost and schedule as a whole without considering in detail the costs of the individual activities that compose the process. A holistic model usually estimates the cost and schedule for activities that compose the overall development by applying a percentage against the overall cost or schedule. These percentages differ for each activity and often vary from one application to another. The most common of the holistic cost and schedule models in the public domain today is COCOMO (Boehm 1981).

6.2.1 The Constructive Cost Model

There are three basic modes in the COCOMO model: organic, semidetached, and embedded. There are also three principal levels of the COCOMO model: basic, intermediate, and detailed. In addition, there is an Ada-COCOMO model. This section describes the three levels as well as the Ada model. Table 6-1 summarizes the basic model.

Table 6-1. Basic Constructive Cost Model Effort and Schedule Equations

Mode	Effort (Cost): $a(KDSI)^b$	Schedule Time: $c(LM)^d$
Organic	$LM = 2.4(KDSI)^{1.05}$	$TDEV = 2.5(LM)^{0.38}$
Semidetached	$LM = 3.0(KDSI)^{1.12}$	$TDEV = 2.5(LM)^{0.35}$
Embedded	$LM = 3.6(KDSI)^{1.20}$	$TDEV = 2.5(LM)^{0.32}$

6.2.1.1 Basic Constructive Cost Model

In Table 6-1, LM is labor months, KDSI is thousands of delivered source instructions, and TDEV is development time in months. Delivered source instructions include a count of physical source statements for new and reused code but exclude undelivered support software such as test drivers. KDSI includes format and data declarations and excludes comments and unmodified utility software. Total KDSI is equal to total KSLOC for the same code categories as described. The LM estimated by the basic COCOMO effort-estimating equation includes the direct effort for product (top-level) software design through integration and acceptance test. Project management, program librarians, documentation effort, quality assurance, and configuration management are included, but personnel, computer center, clerical, facilities, and higher management effort are not included.

The basic effort or cost model in Table 6-1 is of the form:

$$LM = a(KDSI)^b$$

where a is 2.4 and b is 1.05 for the organic equation in Table 6-1. You then compute TDEV as a function of LM. For the organic example in Table 6-1, a is 2.5 and b is 0.38. You can find values for a and b in Table 6-1. Otherwise, you can use regression techniques to calculate them based on data from your enterprise software experience database.

The organic mode applies to a small- to medium-sized product development in a familiar in-house development environment. The embedded mode represents a tightly constrained software product development situation where the product must operate in a complex hardware/software, interactive, procedure-driven system. Military mission-critical and real-time command and control software and air traffic control systems are examples of such products. The semidetached mode is intermediate between organic and embedded, i.e., a development environment with both experienced and inexperienced personnel with an intermediate level of experience relative to the systems under development.

(Boehm 1981) gives the distribution of effort and schedule for the main phases of development for each mode and for a variety of development system sizes. As an example, Table 6-2 shows these distributions for a very large system of 512 KDSI for the embedded mode. See (Boehm 1981, 99) for data on systems of other sizes.

Table 6-2. Embedded Mode Activity and Schedule Distribution

Phase	Percent Effort	Percent Schedule Time
Product design	18	38
Detailed design	24	16
Code and unit test	24	16
Integration test	34	30
Total	100	100

6.2.1.2 Intermediate Constructive Cost Model

The COCOMO intermediate model expands the basic model to include cost multiplier factors which modify the cost estimates (LM) developed from the basic models shown in Table 6-1. Table 6-3 presents these cost multipliers. You can use your subjective judgment to select a ratings level. More detailed definitions of the ratings levels are in (Boehm 1981) if you need them.

Table 6-3. Intermediate Model Effort Multipliers

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	0.75	0.88	1.00	1.15	1.40	
DATA Database size		0.94	1.00	1.08	1.16	
CPLX Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual storage volatility		0.87	1.00	1.15	1.30	
TURN Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	0.86	0.71	
AEXP Applications experience	1.29	1.13	1.00	0.91	0.82	
PCAP Programmer capability	1.42	1.17	1.00	0.86	0.70	
VEXP Virtual machine experience	1.21	1.10	1.00	0.90		
LEXP Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
TOOL Use of software tools	1.24	1.10	1.00	0.91	0.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

The general formula for cost estimation within the COCOMO model is:

$$LM = a(KDSI)^b \cdot \prod_{i=1}^{n} M_i$$

where M_i is the selected cost multiplier from Table 6-3.

Using Tables 6-1 and 6-3, the cost of developing a 512 KDSI (very large), semidetached software product with a high product complexity, low programmer capability, very low virtual machine experience, and very low use of modern software practices (all other factors nominal) would be:

$$3.0(512)^{1.12}(1.15)(1.17)(1.21)(1.24) = 6{,}555 \text{ LM}$$

6.2.1.3 Detailed Constructive Cost Model

A detailed COCOMO model exists with cost multiplier factors for subsystem costs and schedules. These cost multipliers are not totally independent of each other, although they are applied under the COCOMO model as though they were. There is a cost effect overlap between any two and among any group of COCOMO cost multipliers. Where you apply only a very small group of cost multipliers (say, two or three), the effect of this cost overlap is minimal. But if you use a large number of multipliers, the cumulative effect of the overlap may be serious. The estimated costs may be too high or too low.

6.2.1.4 Reuse

You can also use the COCOMO model to estimate development costs in situations involving the reuse of code or the adaptation of modified code in a new application as well as to estimate maintenance costs. You calculate equivalent delivered source instructions (EDSI) as shown in the example presented in this section. EDSI is actually the weighted sum of the constituent percent changes in each of the major phases of development, where the weights are the costs for each of these phases. EDSI is somewhat different from ESLOC (defined in Section 4.2). ESLOC is the weighted size of new and reused code. Table 6-4 gives the quantities to be applied in calculating the EDSI for various reuse and adaptive situations.

Table 6-4. Adaptive Quantities by Phase for Equivalent Delivered Source Instructions

	Adaptive Percent Changes			
Adjustment Factor Components	**Simple Conversion**	**Complex Conversion**	**Extensive Conversion**	**Component Conversion**
DMA Design modification	0	15	35	5
CMA Code modification	15	30	60	15
IMA Integration modification	5	20	140	25

The application adjustment factor (AAF) is defined as:

$$AAF = 0.40(DMA) + 0.30(CMA) + 0.30(IMA)$$

where DMA is the percentage of the adapted software's design that is modified, CMA is the percentage of the adapted software's code that is modified, and IMA is the percentage of effort

required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

The EDSI is defined as:

$$EDSI = ADSI(AAF/100)$$

where ADSI is the number of delivered source instructions (statements) adapted from existing software.

Thus, for a simple conversion (i.e., a simple reuse application) involving 500 statements, in which there is 0 percent change in design, 15 percent change in coding, and 5 percent change in testing:

$$AAF = 0.4(0) + 0.3(15) + 0.3(5) = 6.0 \text{ and } EDSI = 500(0.06) = 30.0$$

These COCOMO models were derived from effort data developed during application of the waterfall model (i.e., sequential projects), as described in Section 6.3. The KDSI and EDSI models assume known and stable requirements. The EDSI model accounts for using only existing code and does not cover the use of existing domain and design information, including the effort to locate and evaluate reusable code.

(Balda and Gustafson 1990) present a COCOMO-based reuse model that overcomes these difficulties. The model is:

$$LM = aN_1^b + 20\gamma aN_2^b + \gamma aN_3^b$$

where a and b are the multipliers and the exponents, respectively, from Table 6-1, and γ (gamma) is the cost ratio of developing a component to be reusable to the cost of developing a component to be unique. The range of γ is from 0.06 to 0.24, so for real-time command and control software, γ will have a value of about 0.2, and for MIS software, γ will have a value of about 0.1. N_1 is the number of unique KDSI developed, N_2 is the number of KDSI developed to be reusable, and N_3 is the number of unchanged, reused KDSI.

(Balda and Gustafson 1990) also present a COCOMO-based prototype cost model for the evolutionary spiral process (see Section 6.3) of the form:

$$LM = aP^b + aI^b + aT^b$$

where a and b are the multipliers and exponents, respectively, from the effort column of Table 6-1. As with the basic COCOMO model, you must select the proper software development mode. P is the SLOC developed for the initial prototype. I is the number of SLOC developed during iterations of the process and is the total of SLOC added, removed, and modified. T is the number of SLOC developed specifically to convert the software to a deliverable product. (This prototype cost model is considered to be an initial model.)

6.2.1.5 Ada Process Model

The Ada Process Model, presented in (Boehm 1987), incorporates the use of Ada, rapid prototyping, risk management, the spiral model, and certain modern software development practices (as shown in Table 6-3 and in [Boehm 1987]) into a model which you can express as a tailored version of

DOD-STD-2176A. The Ada version of COCOMO, which incorporates the effects of the use of Ada and the Ada Process Model is:

$$LM = 2.8(KDSI)^{1.04 + \sum_{i=1}^{4} W_i}$$

where the weights W_i are given in Table 6-5.

Table 6-5. Weights for the Ada-Constructive Cost Model

Weights W_i	Experience With Ada Process Model	Design Thoroughness at PDR: Specifications Compiled	Risks Eliminated by PDR	Requirements Volatility During Development
0.00	Successful on >1 mission-critical project	Fully (100%)	Fully (100%)	No changes
0.01	Successful on 1 mission-critical project	Mostly (90%)	Mostly (90%)	Small noncritical changes
0.02	General familiarity with practices	Generally (75%)	Generally (75%)	Frequent noncritical changes
0.03	Some familiarity with practices	Often (60%)	Often (60%)	Occasional moderate changes
0.04	Little familiarity with practices	Some (40%)	Some (40%)	Frequent moderate changes
0.05	No familiarity with practices	Little (20%)	Little (20%)	Many large changes

Using the guidance in Table 6-5, a software development environment with no familiarity with the Ada Process Model or with the modern software practices of design thoroughness and risk management in an environment of changing requirements would rate a value of 0.05 in all four categories of software practice. Then the Ada-COCOMO estimating equation for 512 KDSI would be:

$$LM = 2.8(512)^{1.24} = 6,407$$

COCOMO is probably the most widely employed cost estimating model. If you do not have any productivity data of your own, then you can use the figures embedded in COCOMO by Boehm. Otherwise, you should use parameters derived from your enterprise's experience.

Table 6-6 gives some values of LM from the Ada-COCOMO model with the associated productivities.

Table 6-6. Sample Effort and Productivities for the Ada-Constructive Cost Model

Sum W_i	100 KDSI		500 KDSI	
	LM	SLOC/LM	LM	SLOC/LM
0.00	336	298	1,795	279
0.04	405	247	2,302	217
0.08	487	205	2,951	169
0.12	585	171	3,784	132
0.16	703	142	4,852	103
0.20	846	118	6,221	80

6.2.2 The Software Life-Cycle Model

The Putnam model or Software Life-Cycle Model (*SLIM*) (Putnam 1978) is a widely used, proprietary, holistic model. The model is:

$$S = C * K^p * t_d^q$$

where S is the software system size in SLOC (excluding comments) or ESLOC when reused code is involved (see Section 4). C is the technology constant which represents both the sophistication of the development environment and the degree of difficulty of software product development. K is the development effort in labor years, and t_d is the development time in years. (Development effort is 40 percent of the total life-cycle effort including maintenance.) The parameter K covers all development activities from design through installation of the software system.

C reflects both the nature of the process to be used in developing and maintaining the software product and the complexity of the intended product. The parameter C will range in value from 1,000 to 5,000 for SEI process maturity level 1 development environments to 40,000 to 75,000 for SEI process maturity level 4 or 5 environments. The effort of developing the difficult, real-time, command and control, embedded software products has the effect of lowering the value of C in all but the most sophisticated development environments.

The parameters p and q were given values 1/3 and 4/3, respectively, by Putnam. (Gaffney 1983) calculates the values of p and q to be 0.6288 and 0.5555, respectively, based on development data from an environment which produces real-time command and control software. You should use the latter set of values, since it is based on well-defined and controlled data from aerospace and real-time command and control software development projects and since this set produces more reasonable results.

As described in Section 7, the Putnam model is useful for testing the reasonableness of the values of K, t_d, and S. For example, with size (S) established, you can submit a value of effort (K) to the model to see if the resulting value of schedule (t_d) is reasonable.

A significant aspect of the SLIM model equation is that you can use it as the basis for developing relationships to make tradeoffs among schedule, size, and effort. Section 7 presents equations developed from the SLIM equation that you can use for this purpose. For example, you can submit the development schedule (t_d) and the size (S) and see how much effort is required (K). Holding the size S

constant, you can increase or decrease K and compute the effect on t_d. This tradeoff method should lead to the right combination of values of S, K, and t_d to suit the specific development situation. You should expect to do several tradeoff exercises between schedule and effort until you obtain an optimum pair of values.

6.2.3 The Cooperative Programming Model

The Cooperative Programming Model (COPMO) is another type of holistic model (Conte, Dunsmore, and Shen 1986). COPMO is designed to reflect the effect of development team size on effort. The effect of team size on development effort is especially strong in large projects where the complex nature of the software causes communication among the technical staff and among the management to be difficult and thus can cause costs to rise dramatically. For large, complex projects, cooperation and communication must be effective to hold costs down.

The COPMO model is:

$$E = E_p + E_c$$

where E_p is the effort in LM for programming and technical development of the software product and E_c is the effort in LM for communication among the development staff. The programming effort is given by:

$$E_p = e + f * S$$

where S is the software size in KSLOC. The communications effort is given by:

$$E_c = g * L^h$$

where L is the average staffing level in LM per month over the duration of software development.

Some parameter values empirically derived from actual development data are $e = 48.0$, $f = 0.33$, $g = 2.02$, and $h = 1.67$. (Conte, Dunsmore, and Shen 1986) gives similar values.

6.2.4 How to Apply Holistic Models

If your organization has little or no accumulated software development experience, you can apply holistic models such as COCOMO, SLIM, and COPMO using the parametric values presented here. As you gain and record software experience, you should modify or customize the holistic model parameter values to better represent your own software process. Regardless of the degree of development of the enterprise estimating process, you can use the holistic models as a cross-check on each other. You should make estimates for any situation using two or more models, and you should compare their results. If the results differ by more than 10 to 20 percent (either cost or schedule time), then you should closely examine your assumptions and data about the software project in question.

As an example of cross-checking, suppose that you use the basic COCOMO semidetached model to estimate the cost of developing 660 KDSI. The estimate is $3.0(660)^{1.12} = 4{,}315.3$ LM. If you use SLIM to calculate the effort involved in developing 660 SLOC in 3.5 years with a technology constant of 10,000, the result involves solving the following relation for K:

$$660{,}000 = 10{,}000 * K^{0.6288} * 3.5^{0.5555}$$

The result is 258.8 labor years, or 3,105.6 LM, a value which does not compare well to the COCOMO-generated value. If you remember that the COCOMO value includes quality assurance, configuration management, and software builds which are an additional 19 percent (see Section 6.8), then the COCOMO value becomes 4,315.3/1.19 = 3,626.3 LM. This value is only 14 percent different from the SLIM value, so you can consider both values essentially the same.

In general, when two estimates do not compare in a cross-check, you should look for differences in the definition of work to be done. Since KSLOC and KDSI are essentially the same, the difference is in the range of activities implied by each model.

If your organization is at SEI process maturity levels 1 and 2, you should use holistic models for cost and schedule estimation and for cross-checking purposes. You should initiate an enterprise software experience database to preserve cost experience, and you should use the data to customize the holistic model parameter values to your development environment.

6.3 ACTIVITY-BASED MODELS

This section describes activity-based models and how you can apply them to do estimates for your projects. Holistic models represent a great improvement over the ad hoc cost estimation methods of the past, but they are not as accurate as the activity-based methods. To implement activity-based models, you need to accumulate software process and product information, preferably recorded in a formal database. Even when your software development process is mature enough to implement activity-based models with customized cost data, you can still use the holistic models as an approximate cross-check on the estimated overall software project costs.

6.3.1 Activity-Based Model and Process Maturity Levels

At SEI process maturity level 3, your software development organization will have sufficient experience data about its development process to use activity-based models. At this process maturity level and above, you will know all of the activities that must occur in your development process and the labor rates (or productivities) that characterize each of these activities.

6.3.2 The General Form of the Activity-Based Model

You can think of the development process for a software product as consisting of a set of subprocesses (called phases in this guidebook). The subprocesses are composed of activities that have been selected from a "menu" of activities that you are to apply in some order appropriate to that product. Each of these activities is constructed according to a specific paradigm or set of rules, so as to answer the following questions:

- What is the nature of the input to the process?
- What is the nature of the output from the process?
- What is the nature of the transformation from input to output?
- How is it known when the process is completed?

This is the general form of the activity-based model, and this guidebook defines process metrics to provide quantitative answers to these questions.

An activity-based cost model considers all of the activities in the development life cycle, such as requirements, design, and implementation, and assigns a unit cost to each of them. You should base these unit costs on data from your enterprise's experience. If you do not have such data, you could employ figures obtained from Tables 6-7, 6-8, and 6-9 (see pages 6-13, 6-15, and 6-17). Use the data for the activities that you plan to employ in the development process for the product whose costs you are estimating. Typically, your project will not use all of the "repertoire" of activities. For example, if you are developing a new version of an existing system, you might not have any high-level design in your development process. You usually measure unit costs in LM/KSLOC or LH/SLOC, although other metrics may be appropriate under certain circumstances. You should define your software development life cycle in terms of known and measurable activities. Tables 6-8 and 6-9 contain examples of activity-based models. Table 6-8 lists the activities in the waterfall model as defined in DOD-STD-1521B (Department of Defense 1985) and as illustrated in DOD-STD-2167A (Department of Defense 1988) with some actual (empirically determined) labor rates derived from data collected during the development of 25 large (over 500 KSLOC), real-time command and control software systems. Table 6-9, presented in Section 6.3.4, lists the activities in the spiral model (Boehm 1988) with some labor rates that have been determined by inference from the waterfall model. The LM in the Table 6-8 and Table 6-9 metrics are based on 167 hours.

The activity-based model estimates overall costs in LM or LH and is based on individual labor rates for each activity in the development process and on the size of the software product to be produced. The general form of the activity-based model is:

$$TLM = \sum_{i=1}^{n}(LM/KSLOC)_{i,added} * KSLOC_{added} + \sum_{i=1}^{n}(LM/KSLOC)_{i,modified} * KSLOC_{modified}$$

$$+ \sum_{i=1}^{n}(LM/KSLOC)_{i,reused} * KSLOC_{reused} + \sum_{i=1}^{n}(LM/KSLOC)_{i,removed} * KSLOC_{removed}$$

where TLM indicates the total effort in LM for all n activities, and $(LM/KSLOC)_{i,j}$ indicates a labor rate in LM/KSLOC for activity i and code category j.

This equation is stated in terms of the code types discussed in Section 4 and is the general form for all activity-based models, including the waterfall and the (evolutionary) spiral models. You can simplify this general form by weighting the modified code the same as added code (since both are costed at the same rate) and then combining the added and modified code into the new code category. You also usually weight the removed code 0.0 (i.e., no cost for removing code). This procedure reduces the above equation to:

$$TLM = \sum_{i=1}^{n}(LM/KSLOC)_{i,new} * KSLOC_{new} + \sum_{i=1}^{n}(LM/KSLOC)_{i,reused} * KSLOC_{reused}$$

where TLM indicates the total effort in LM for all n activities and $(LM/KSLOC)_{i,j}$ indicates a labor rate in LM/KSLOC for activity i and code category j. You can use either actual or estimated size data.

An alternative form of the previous cost (effort) estimating equation employs the categories "new" and "reused" code only. It is the recommended format to employ.

$$TLH = \sum_{i=1}^{n}(LH/SLOC)_{i,new} * SLOC_{new} + \sum_{i=1}^{n}(LH/SLOC)_{i,reused} * SLOC_{reused}$$

where TLH indicates the total effort in LH for all n activities and $(LH/SLOC)_{i,j}$ indicates a labor rate in LH/SLOC for activity i and code category j. You can use either actual or estimated size data.

The labor rates for modified code are considered to be the same as added code, and the labor rates for removed code are considered to be zero, so that only new (added plus modified) and reused code appear in the equation. In other words, you combine the costs for the code types shown above and in Section 4 into just two code categories, new and reused. This categorization, in effect, states that any new software product is composed of new and reused code in varying proportions.

The labor rates in Table 6-8 were derived from actual cost and size data from embedded software development projects in the aerospace industry. (Cruickshank and Lesser 1982) present similar rates for a related sample of activity-based models.

To use the activity-based model approach effectively, you must first decompose your project software development process into its constituent activities and then map these activities to the activities listed in Tables 6-8 or 6-9. Your software development process may not include all of the activities listed in these tables. It may be that your software development organization is not responsible for the range of activities shown in the tables, or perhaps some activities have been "tailored" out of the process requirements. When you have compiled a list of activities that accurately describes your software development process, then you can select labor rate values from the table guidelines, or you can derive new values based on data from your enterprise software experience database. Section 4.3 shows how to estimate the labor rates for the various activities from which the software development process is composed.

Alternatively, you may sample Tables 6-8 or 6-9 to create a software development process model which represents the actual process. This method is especially useful when the development is governed by DOD-STD-1521B and DOD-STD-2167A. Think of the set of activities in these tables as "menus" from which you can select the activities to actually use in your software development process.

The labor rates in Tables 6-8 and 6-9 include the costs (about 13 percent) for first-line management and applicable second-line management (Cruickshank 1988). Additional costs such as quality assurance, configuration management, and program management are covered in Section 6.8.

As an example, suppose that you are to develop software consisting of 450 new KSLOC and 710 reused KSLOC using a waterfall method and that you are to perform the activities of preliminary design, detailed design, coding and CSU testing, CSC integration testing, CSCI testing, and error correction. Suppose further that line management determines that the new requirements are insufficiently defined and that, therefore, design for new code will cost 20 percent additional, i.e., 20 percent above the base labor rates for design. You should derive such modifier factors as the 20 percent shown here on the basis of a systematic subjective evaluation based on past experience from an aerospace (or similar) organization. Also, suppose that the programmers involved lack sufficient experience in the language selected so that coding and unit testing will cost an additional 10 percent, again based on a systematic, subjective evaluation. Using the labor rates in Table 6-8 as base labor rates, you then determine the cost calculations as shown in the Table 6-7 worksheet.

Table 6-7. Worksheet Cost Calculations for an Activity-Based Model

Project Name	WATERFALL		**Date**	6/1/91
CSCI/Product Name	EXAMPLE		**Language**	JOVIAL
Labor Rate Measurement (LM/KSLOC, LH/SLOC, etc.)			LM/KSLOC	
New Code Size	450	**Reused Code Size**		710
	New Code			
Phase/Activity	**Base Labor Rate**	**Modifier**	**Estimator Labor Rate**	**Cost (LM)**
Preliminary design and documentation	0.59	1.20	0.71	319.5
Detailed design and documentation	0.89	1.20	1.07	481.5
Code and CSU test	2.21	1.10	2.43	1,093.5
CSC integration test	0.74	1.00	0.74	333.0
Error correction	0.41	1.00	0.41	184.5
CSCI test	0.73	1.00	0.73	328.5
Error correction	0.28	1.00	0.28	126.0
Total	5.85		6.37	2,866.5
	Reused Code			
Phase/Activity	**Base Labor Rate**	**Modifier**	**Estimator Labor Rate**	**Cost (LM)**
Preliminary design and documentation	0.035	1.00	0.035	24.8
Detailed design and documentation	0.055	1.00	0.055	39.0
Code and CSU test	0.120	0.00	0.000	0.0
CSC integration test	0.200	1.00	0.200	142.0
Error correction	0.060	1.00	0.060	42.6
CSCI test	0.150	1.00	0.150	106.5
Error correction	0.040	1.00	0.040	28.4
Total	0.660		0.540	383.3
Overall Total LM				3,249.8

Using the total costs from Table 6-7, the new code productivity is estimated to be (450,000/2866.5) = 160.0 SLOC/LM over all of the activities in the development process that you intend to use in creating the new software product. Since KESLOC is 450 + (0.30)(710) = 663, the overall productivity is (663,000/3,249.8) = 204.0 ESLOC/LM.

You should view Table 6-7 not only as an example of cost calculations for a activity-based model, but also as worksheet designed for the convenience of a cost estimator.

You should view the labor rates in Tables 6-8 and 6-9 as an example or as guidance based on some specific experience in the aerospace industry with developing real-time command and control software. Your experience or judgment as a cost estimator may lead to a modification of some or all of these rates—up or down. For example, in a situation where many CSCs are to be integrated together, you may want to increase the labor rate for CSC integration test to account for additional complexity.

The same is true of the functional testing of the software system where you integrate many CSCIs in CSCI test.

The optimum estimation situation, both for estimating cost and software size, is one in which your enterprise creates, updates, and maintains a database of actual software and documentation sizes and costs recorded by software development activity. Such a database, which will exist when your enterprise is at SEI process maturity levels 3 or 4, allows you to create unit costs (labor rates) and overall cost estimates tailored to the enterprise software development environment.

If a set of standard or guideline labor rates based on such a database, as suggested above, is not available, then you can use the COCOMO detailed model, specifically the modern programming practices (MODP) effort multipliers shown in (Boehm 1981, Table 27-1). Using this scheme, you can modify the baseline labor rate for each activity from requirements through integration and test by a multiplier factor to adjust for the degree of adherence to modern programming practices. (Boehm 1981) gives a set of factors to be used in this case.

If your organization is at SEI process maturity levels 3 through 5, you should use your enterprise software experience database to derive labor rates for each of the activities in each defined software development process in use. With these labor rates, you should use activity-based models for cost estimation.

6.3.3 The Waterfall Model

The waterfall model of the software development process presented in Table 6-8 is a stagewise model in that it is composed of a succession of activities. The waterfall model incorporates the possibility of iteration between successive major steps. This feedback mechanism is allowed only between successive steps so that if, for example, you encounter an error in coding, you can only feed it back to the detailed design step (phase) and not directly to the preliminary design step (phase). If you then feed back this error to the preliminary design step (phase), a costly redesign effort may be necessary.

The activities in the waterfall model, as presented in Table 6-8, are specified in DOD-STD-1521B and illustrated in DOD-STD-2167A. The labor rates have been derived from the experience of an aerospace software organization that developed real-time command and control software in both pre-2167A and post-2167A time frames. The labor rate data has been verified by comparing estimates that used the data with actual costs.

The labor rates in Table 6-8 are for new and reused code from actual project data. The ratio of the new code rate to the reused code rate differs from one activity to the next activity, and this condition is to be expected since the software development activities differ widely in character. Section 4 shows how you can derive such rates.

6.3.4 The Evolutionary Spiral Model

The evolutionary development model of the software development process focuses on delivering a sequence of versions of a functioning software system before delivering the final system. With each delivery, you incorporate changes resulting from the development and application experience into the specifications for the next version to be delivered. The final system benefits from the continual feedback from the user and from the developer.

Table 6-8. Waterfall Model Activities and Labor Rates

Phase	Activity/Product (DOD-STD-1521B and DOD-STD-2167A)	Unit Cost (LM/KSLOC)	
		New	Reused
System requirements analysis	System/segment design document	0.31	0.020
	Software development plan	0.13	0.010
	Preliminary software requirements specification	0.25	0.020
	Preliminary interface requirements specification	0.04	0.002
Software requirements analysis	Software requirements specification	0.39	0.030
	Interface requirements specification	0.04	0.002
Preliminary design	Software design document—preliminary design including design reviews	0.52	0.030
	Preliminary interface design document	0.07	0.005
	Software test plan	0.13	0.005
	CSC test requirements	0.01	0.000
Detailed design	Software design document—detailed design including design reviews	0.82	0.050
	Interface design document	0.07	0.005
	CSU test requirements and test cases	0.01	0.000
	CSC test cases	0.02	0.000
	Contents of CSU and CSC software development files	0.00	0.000
	Software test description—test cases	0.04	0.006
Coding and CSU testing	Implement source code including code inspections	1.48	0.070
	CSU test procedures	0.03	0.000
	CSU testing	0.73	0.050
	CSC test procedures	0.25	0.030
	Contents of CSU and CSC software development files	0.00	0.000
CSC integration and testing	CSC integration testing	0.74	0.200
	Software test description—formal test procedures	0.10	0.020
	Updated source code—error correction	0.10	0.010
	Contents of updated software development files	0.00	0.000
CSCI testing	CSCI testing including acceptance testing	0.73	0.150
	Software test report	0.01	0.000
	Updated source code—error correction	0.05	0.010

The basic spiral model (Boehm 1988) is an alternative to the waterfall model. The spiral model provides a disciplined, yet flexible framework designed to overcome the more rigid, linear progression of steps inherent in the stagewise and waterfall models. Processes based on the spiral model permit a more product-driven approach to developing a software system. They incorporate such techniques as prototyping, reuse, and risk analysis. A process based on the spiral model is said to be risk-driven. It is important that you reduce the basic spiral model to an activity-based model since you must

express the various versions and applications of the spiral model, such as the Evolutionary Spiral Model, in terms of the individual tasks that compose that version or application. Otherwise, the model would lack sufficient flexibility.

The Evolutionary Spiral Model is a combination of the evolutionary and the spiral models (Software Productivity Consortium 1991). It incorporates prototyping in the form of successive versions of the final system and risk analysis involving the feedback of experience to test whether the next delivery should constitute the final system.

Table 6-9 is an example of an activity-based representation of the Evolutionary Spiral Model. A cursory examination of this table shows that many activities seem to cut across the phases or iterations. For example, requirements-related efforts occur in the first three phases, protoyping occurs in the first four phases, and design-related efforts in phases 3 and 4. The iterative nature of the evolutionary spiral process causes this spreading of activities. It concentrates on subactivities unique to each phase. It covers requirements analysis in one phase and requirements planning in another. But, just as with all activity-based models, the total cost is the sum of the LM for each activity across the various phases in which it occurs. In the literature on spiral models, phases are also known as iterations or cycles.

The Evolutionary Spiral Model does not assign individual development activities (e.g., requirements analysis) to specific phases of the spiral model. In fact, you can repeat a development activity in several phases after the phase in which it first appears. Within the evolutionary spiral "envelope" or general class of models, there can be many phase/activity combinations. Each such set of phase/activity combinations is a representation (or example) of the Evolutionary Spiral Model. A specific representation of the Evolutionary Spiral Model is (known as) an evolutionary spiral process. Table 6-9 presents such an example.

The new and reused code labor rates were derived from corresponding activity rates in the waterfall model of Table 6-8. Other examples of the evolutionary spiral process will require different mappings.

6.3.5 Pooling Estimates

It is important for you to be familiar with the software development process for which you are making the cost estimate. Knowledge of the process means detailed knowledge of the activities that compose the process. You may use Tables 6-8 and 6-9 as "menus" from which you can select the labor rates that correspond to the activities in the development process. You can modify the corresponding (guideline) rates to suit the situation. If no activity in these tables corresponds to an activity in your intended development process, then you must find some other way to estimate the corresponding labor rate. Often it helps to find an analogous activity in a past development project and to use the associated labor rate data. This selection process is known as modeling the enterprise software development process.

It is often helpful to make a cost estimate using an activity-based model and then cross-check the estimate by using one or more of the holistic models. If the estimates do not differ by more than 10 percent (at most 20 percent), then you may conclude that the results from all the estimating techniques and models are giving a consistent result. In the event that this desirable state of affairs does not exist (i.e., that some of the models give significantly different cost estimates), then you must analyze the data and the models themselves to find an explanation. It may be that some of the models are not suitable for application to the situation at hand. Experience will show which cost estimating models and techniques are best for a given software development environment.

Table 6-9. Example of Evolutionary Spiral Model Activities and Labor Rates

Phase/Iteration	Activity/Product	Unit Cost (LM/KSLOC)	
		New	Reused
1A	Requirements analysis	0.28	0.020
	Prototype 1	0.11	0.020
	Determine concept of operation	0.14	0.000
	Requirements plan	0.05	0.000
	Life-cycle plan	0.08	0.000
1B	Determine and evaluate objectives, alternatives, constraints	0.01	0.000
	Verify next level process plans	0.01	0.000
	Risk analysis and resolution	0.02	0.000
2A	Prototype 2—simulations	0.22	0.020
	Software requirements	0.29	0.020
	Requirements validation	0.03	0.000
	Software development plan	0.13	0.010
2B	Determine and evaluate objectives, alternatives, constraints	0.01	0.000
	Verify next level process plans	0.01	0.000
	Risk analysis and resolution	0.02	0.000
3A	Prototype 3—models	0.77	0.020
	Detailed requirements	0.43	0.030
	Software product (top-level) design	0.80	0.030
	Design validation and verification	0.09	0.003
	Integration and test plan	0.13	0.020
3B	Determine and evaluate objectives, alternatives, constraints	0.01	0.000
	Verify next level process plans	0.01	0.000
	Risk analysis and resolution	0.02	0.000
4A	Operational prototype—benchmarks	0.55	0.020
	Detailed design	0.89	0.060
	Code and unit test	0.77	0.020
4B	Integration test (CSC integration test and CSCI test)	1.40	0.210
	Acceptance test	0.57	0.085
5	Installation	0.73	0.110

6.3.6 Point and Interval Estimates of Labor Rates

When used with a single estimate of size, the labor rates or unit costs in Tables 6-8 and 6-9 will produce a point estimate of cost, i.e., a single estimate of cost with no information about variation or the implied statistical distribution. You should try to produce an interval estimate—a range of possible cost values with the statistical distribution of values—as well as a point estimate. One possible way to do this is to estimate costs based on a sample of estimated software sizes, with each of the size estimates

produced by a different method. Section 5 presents several sizing methods that are useful in this context. Another possible method is to generate a distribution of sizes by assigning probabilities to a range of size values. Section 6.9 gives an example of this method and shows you how to compute an estimate of cost risk.

6.3.7 The Cost Effect of a Higher Order Language

Using a higher order language (*HOL*) has an effect on software development costs. (Gaffney 1986) establishes that the cost ratio of development in an HOL to development in assembly-level language is:

$$M = \frac{1 + \frac{C}{X}}{1 + C}$$

where C is the ratio of the (estimated) costs of software design and system testing costs to the software coding and unit test costs, and X is the ratio of the size of the software function or product in assembly-level code to the size in HOL, i.e., the inverse of the language level. (Gaffney 1986) gives a value of 0.59 for C and 3.0 for X for one development situation. You can use M as a multiplier on software development costs in situations where you wish to account for the effect of the use of an HOL on software development costs.

6.3.8 The Cost Effect of Software Product Size

You should check for the effect that the software product size has on unit costs. As software product size increases beyond some given size, the unit costs generally rise. (Gaffney and Werling 1990) give a general relationship of unit cost as a function of size:

$$U = y(KSLOC)^{z-1}$$

where U is unit cost in LM/KSLOC. Some values (Gaffney and Werling 1990) for aerospace software are $y = 5.091$ and $z = 1.0559$. The relationship also holds if you measure unit cost in LH/SLOC and size in SLOC, but you would have to calculate revised parameter values accordingly.

This relationship between unit cost and size was derived using new code development data. You should explore the sensitivity of your project unit costs by inserting several values of the size in KSLOC somewhat above the current estimate of size into this equation and gauging the effects on U. If U increases by more than 10 percent, there is probably a risk exposure.

6.4 HOW TO ESTIMATE DOCUMENTATION COSTS

The methods, equations, and parameter values shown in Sections 6.4 through 6.8 are based on the analysis of the enterprise software experience database (actual costs) of an aerospace developer of real-time command and control software. The database contained over 6 million source statements (new and reused) developed with more than 1,000 labor years of effort over 40 major software products in the years 1976 to 1988 (Cruickshank 1984, revised).

(Hancock 1982) and (Cruickshank 1984) give formulas for estimating the number of pages of documentation from program size (SLOC) information and for estimating the documentation effort (LM) from the estimated pages. Table 6-10 provides formulas that you can use for estimating the number of pages of documentation from the software size. In Table 6-10, P stands for pages and K stands for KSLOC.

Table 6-10. Estimating Pages From Software System Size

Document	Estimating Formula	KSLOC Range
System and software requirements	$P = 10.0 K - 0.073 K^2$	Below 110
	$P = 2.0 K$	Above 110
Preliminary software design, preliminary interface design	$P = 16.0 K - 0.061 K^2$	Below 205
	$P = 3.5 K$	Above 205
Detailed software design, interface design	$P = 16.0 K - 0.085 K^2$	Below 165
	$P = 2.0 K$	Above 165
Test plans and requirements	$P = 7.0 K - 0.034 K^2$	Below 175
	$P = 1.0 K$	Above 175
Test procedures and cases	$P = 24.0 K - 0.140 K^2$	Below 150
	$P = 3.0 K$	Above 150
User and operator manuals	$P = 5.0 K - 0.020 K^2$	Below 200
	$P = 1.0 K$	Above 200

The relationships shown in Table 6-10 came from data generated before the publication of DOD-STD-2167A, but these relationships have been extensively tested and used in estimating situations since the publication of DOD-STD-2167A.

Table 6-11 provides the formulas for estimating the documentation effort in LH and LM from the estimated number of pages. The effort shown here includes the analysis time as well as writing a first draft. In Table 6-11, P stands for the number of pages and KP stands for 1,000 pages, so that the effort in LH is on a per page basis and the effort in LM is on a per thousand page basis.

Table 6-11. Estimating Documentation Effort From Document Size

Document	Estimating Formula (LH)	Estimating Formula (LM)
System and software requirements, preliminary design documents	$LH = 17.6 P$	$LM = 105.6 KP$
Detailed design documents, test plans and requirements	$LH = 13.3 P$	$LM = 79.8 KP$
Test procedures and cases	$LH = 7.1 P$	$LM = 42.6 KP$
User and operator manuals	$LH = 3.7 P$	$LM = 22.2 KP$

Each of the estimates produced by the use of the formulas in Tables 6-10 and 6-11 has an associated "impact factor" which does not appear in the formulas in Table 6-11. The impact factor says that it is not enough to estimate the technical effort or cost by the estimating formulas. For example, the real cost of documentation is the cost of getting the document completed, which includes not only writing but also reviews and revisions, customer interfacing, and management. Reviews and revisions to the draft and preliminary versions of the document will cost an additional (to the first draft costs) 17 percent, customer interfacing will cost an additional 8 percent, and management will cost an additional 13 percent. So the total impact factor is 1.38, and you must multiply the cost or effort of writing the document by the appropriate value of the impact factor to get the true cost of the document.

Suppose that you are to develop a computer program of 450 KSLOC, and you want to know how much the preliminary and detailed software design document will cost. Using Table 6-10 to estimate the pages involved, the preliminary design documentation will be (3.5)(450) = 1,575 pages, and the detailed design documentation will be (2.0)(450) = 900 pages. Using Table 6-11 to get costs, the preliminary design documentation will cost (105.6)(1.575) = 166.3 LM, and the detailed design documentation will cost (79.8)(0.900) = 71.8 LM. These pages could be a mixture of text, figures, tables, and listings of a design language. When the impact factor is taken into consideration, the total cost would be (166.3 + 71.8)1.38 = 328.6 LM.

You should estimate documentation size and effort separately from the other software development activities. For certain development activities, such as design, where the output is a document (electronic or hard copy), the estimate of documentation costs can serve as a cross-check on the estimate of design activity costs using LM/KSLOC.

6.5 HOW TO ESTIMATE TESTING COSTS

Table 6-12 (Hancock 1982) shows the formulas for estimating testing and problem analysis effort in LH. In Table 6-12, R is the number of testers and analysts involved, and T is the number of test procedure steps.

Table 6-12. Estimating System and Software Testing Effort

Testing	Estimating Formula (LH)	Problem Correction (LH)
System (*HWCI*/CSCI) integration, software (CSC) integration	LH = (0.958 + 0.217 R) T	LH = 1.167 T
Acceptance	LH = (0.125 + 0.033 R) T	LH = 0.333 T
Installation/verification	LH = (0.417 + 0.083 R) T	LH = 0.333 T

The cost of testing also has an impact factor that leads to a true cost beyond the costs of the actual testing. The start-up costs (inefficiencies) add another 5 percent and management adds 13 percent, so that the impact factor is 1.18. Problem analysis also requires management, so the impact factor for this activity is 1.13. The impact factor is not included in the parameter values of Table 6-12.

One test procedure page will contain about six test procedure steps, so you can estimate the number of test procedure steps from a count of the pages in a test procedure document. About 20 test procedure steps can be covered per hour, but you should customize this parameter to your environment.

As an example of estimating testing costs, suppose that 500 KSLOC are to undergo system testing on a target computer. Table 6-10 indicates that (500)(3.0) = 1,500 pages of test procedures are required for this effort. Since one test procedure page contains (nominally) six test steps, an estimated T = 9,000 test procedure steps are involved. Furthermore, suppose that you are to staff three test laboratories for two full shifts at eight testers in each laboratory. This plan means a total of 48 testers will be involved. The calculation of test effort is:

$$(0.958 + 0.217(48))\ 9{,}000 = 102{,}366\ \text{LH} = 612.9\ \text{LM}$$

If this system test effort is considered to be equivalent to CSCI test, then Table 6-8 indicates that the cost is (1.22)(500) = 610 LM, a result remarkably close to the cost estimate above.

6.6 HOW TO CROSS-CHECK ESTIMATES

6.6.1 Design Cost Estimate Example

Suppose you wish to estimate the costs of software design documentation (but not test documentation) for a software project of 400 KSLOC in size. Table 6-8 shows that you require two design documents —the SDD and the IDD. Additionally, you must generate each in preliminary form and final form. Using the new document labor rates from the table, the unit cost of documentation is (0.52+0.07+0.82+0.07) = 1.48 LM/KSLOC and the estimated total cost is (1.48)(400 KSLOC) = 592.0 LM.

To cross-check this estimate, use the documentation models presented in Tables 6-10 and 6-11. From Table 6-10, the preliminary design, detailed design, preliminary interface design, and interface design will generate (3.5)(400) = 1,400 pages for the preliminary SDD and (2.0)(400) = 800 pages for the detailed SDD. Estimate the same number of pages for the preliminary and final IDD. Using Table 6-11, the preliminary SDD will cost (105.6)(1.4) = 147.8 LM, as will the preliminary IDD. The detailed SDD will cost (79.8)(0.8) = 63.8 LM, as will the final IDD, for a total of 423.2 LM. Applying the impact factor, the total cost is (423.2)1.38 = 584.8 LM. This estimate is clearly within 10 percent of the activity-based estimate. You may use either estimate.

6.6.2 Testing Cost Estimate Example

Suppose you wish to estimate the costs of CSC testing and the associated error correction costs for a software development project delivering 400 KSLOC of new code. The activity-based waterfall model in Table 6-8 says that you can estimate the cost of CSC integration testing at (0.74)(400) = 296 LM, and you can estimate the cost of error correction as (0.41)(400) = 164 LM for a total of 460 LM.

Table 6-12 contains the formula for estimating the CSC test and integration effort, but first you must estimate the number of test procedure steps (T) and the number of testers (R). Table 6-10 states that the test plans and test requirements documents will be a total of (1.0+1.0)(400) = 800 pages in length and that the test procedures and cases documentation will be (3.0+3.0)(400) = 2,400 pages in length. With 3,200 pages at 6 test procedure steps per page, you yield an estimated T = 19,200 test procedure steps, and at 20 steps per hour, this testing effort is 960 LH or 120 labor days. A testing team of four should be able to do this testing in 30 labor days which seems reasonable, so R = 4.

So the labor hours involved in actual testing are, from Table 6-12, (0.958+0.217(4))19,200 = 35,059.2 LH or (dividing by 167) 209.9 LM. The effort for problem solving and error correction is (1.167)19,200 = 22,406.4 LH or 134.2 LM. The total LM is 344.1 and, when you apply the testing impact factor of 1.18, the total LM is 406.0.

Comparing the 406.0 LM estimate with the activity-based estimate of 460.0 LM shows that they are within about 12 percent of each other. Being within 20 percent of each other (and actually closer to within 10 percent of each other), they are substantially the same. You can use either estimate or you can average the two estimates.

6.7 TOP-DOWN ESTIMATION OF TOTAL SYSTEM DEVELOPMENT COSTS

Often, projects entail the simultaneous development of computer hardware and software for a new system. (Here system is used in the sense of an interacting group of target computer hardware and

software items developed simultaneously.) This section presents a top-down method for estimating the allocations of cost resources to all of the major aspects of system cost. The methodology is called top-down since the only inputs to the methods are the total amount of resources (in dollars or LM) and the applicable aspects of the developmental system.

The top-down estimating method represents the view that the total system costs (of which software is a part) are proportional to the costs of the major cost drivers such as shown in Table 6-13. You can use estimating algorithms to estimate the costs of each of those cost drivers. In addition, the method represents the costs of all of the major cost drivers as proportional to the software development costs. Table 6-13 shows the proportions and the corresponding estimating algorithms for a developmental system and all of the major cost drivers for an example of a full-scale program involving the simultaneous development of computer hardware and software.

Table 6-13. Top-Down System Development Estimating Models

Program Cost-Driver	Proportion	Algorithm
Software development (*SW*) (including builds and libraries)	0.22	
Computer hardware development (*HW* design and model)	0.14	0.65 SW
Systems engineering (*SE*)	0.10	0.30 (HW + SW) or 0.50 SW
Test and evaluation (*TE*) (software and system testing/validation)	0.11	0.25 (SW + HW + TE) or 0.50 SW
Manufacturing (*MFG*) (full-scale development)	0.12	2 to 5 systems
Product support (logistics)	0.06	0.09 (SW + HW + SE + TE + MFG)
Configuration management, data management, and quality assurance (*CM, DM, QA*)	0.07	0.10 (SW + HW + SE + TE + MFG)
Program management	0.18	Program management, financial management, clerical, cost engineering, measurement
Total	1.00	

All organizations, no matter what their process maturity, should use top-down models at the earliest stages of conceptualization to estimate the allocation of resources (effort) to the general project tasks to be accomplished. You can use the parameter values presented here until your organization accumulates enough experience data to derive its own allocation parameter values.

Suppose that your project is to develop 500 KSLOC of new code with no associated computer hardware development or manufacturing. Starting with the given software proportion of 0.22, the proportion for computer hardware development will be 0.0; systems engineering will be 0.5 SW = 0.11; test and evaluation will be 0.5 SW = 0.11; manufacturing will be 0.0; logistics will be 0.09 of SW + SE + TE = 0.04; CM, DM, and QA will be 0.10 of SW + SE + TE = 0.04; and program management will be 0.18. The proportions add up to 0.69. The adjustment factor will be 1/0.69 = 1.45. You will then use this adjustment factor to adjust each of the proportions up to total 1.00. These adjusted project proportions appear in Table 6-14.

Table 6-14. Top-Down Software System Cost Estimating Example

Program Cost-Driver	Proportion	Cost (LM)
Software development	0.30	2,500
Computer hardware development	0.00	0
Systems engineering	0.16	1,333
Test and evaluation	0.16	1,333
Manufacturing	0.00	0
Product support	0.06	500
CM, DM, QA	0.06	500
Program management	0.26	2,167
Total	1.00	8,333

You can estimate the software size by the methods in Section 5, and you can estimate the software costs by the methods in Sections 6.2 and 6.3. When you have estimated the software development and test costs, all other project costs will be proportional, as above, to the software development costs. The total of these costs will be the total system development costs.

As an example, suppose that it costs 5.00 LM/KSLOC (200 SLOC/LM) for software development planning, preliminary and detailed design (systems engineering will provide the software requirements and that effort will not be estimated in this example), development (coding and unit test), error correction support, CSC integration, and builds and libraries. Also assume that no computer hardware development or manufacturing is required. Then the software costs (for 500 KSLOC) will be (5.00)(500) = 2,500 LM. All other costs will be proportional to the software costs as shown in Table 6-14. The data in Table 6-13 is (revised) data from (Cruickshank 1988).

As a reverse example, suppose that $30,000,000 is available for the development of computer hardware and software and for all of the project tasks shown in Table 6-13. How much software can you develop within this budget for this project? Software activities include design, development, integration test, and builds and libraries.

Assuming $10,000 per LM for labor and burden, 3,000 LM are available for the whole project. Using Table 6-13, 3,000(0.22) = 660 LM will be the software labor budget. At 5.00 LM/KSLOC (as in the previous example) for the above software development activities, you can develop 660/5.00 = 132 KSLOC.

6.8 HOW TO ESTIMATE COSTS OF SUPPORT TO SOFTWARE DEVELOPMENT

Table 6-15 lists the costs of support to software development as additional percentages of cost. For example, if the cost (design through CSC integration test) of a software product were 100 LM, then measurement would cost an additional 2 percent or 2 LM to provide estimation support and historical cost data.

Table 6-15. Percent Additional Cost for Support to Software Development

Activity	Percent Additional Cost
Quality assurance	5–15
Builds, libraries	8
Configuration management	1
Program management including financial management	13–15
Measurement	2
Clerical	2

These percentages indicate costs in addition to the software development costs and do not include support costs for computer hardware development. Measurement activities include cost and size estimating, monitoring software costs, data collection, and reporting. "Front-end" measurement activities such as the initial establishment of an enterprise software experience database and cost proposal activities before the project officially begins are not included. The data in Table 6-15 is (revised) from (Cruickshank and Lesser 1982).

You should estimate the costs of support to software development for every project.

6.9 RISK IN ESTIMATES OF COST

It is important to have a quantitative measure of the variation inherent in an estimate of cost, for this variation defines the risk. The cost estimate plus some multiple of the inherent variation represents the upside cost exposure to the software development project. If this upside potential exposure is so high that it might cause a serious cost overrun, or if it might cause a decrease in the functionality delivered or a decrease in the product quality, then your project schedule is at risk and you would have to increase your cost estimate and its associated budgets. Conversely, if the cost estimate minus some multiple of the inherent variation is too low, then the project is at negative risk and you may assign unnecessary resources to the project, causing productivity to be low and costs to be high. You must be able to quantify this inherent variation to be able to make judgments about the upside and downside exposures.

You may treat an estimate of software cost in two ways—as a fixed quantity or as a sample from a statistical distribution. You calculate the variation inherent in a cost estimate quite differently in each of these cases. This section presents these two methods of modeling and calculating this inherent variation. Throughout this section the inherent variation in a software cost estimate is called "cost risk."

6.9.1 Point Estimates of Cost

The point estimate of cost is the most common type of estimate. In this case, you view the cost estimate as a fixed quantity which is equivalent to a statistical, unbiased point estimate. (You might view a proposed cost as a statistically biased estimate.) In this case, the variance of a cost estimate is defined as a percentage of the unbiased cost estimate.

The variation of a cost estimate (i.e., the percent cost risk) is defined in the case of the unbiased point estimate of cost by the relation:

$$\text{Risk} = \frac{(\text{Expected\_Cost}) - (\text{Dictated\_Cost})}{(\text{Dictated\_Cost})} \cdot 100$$

The "expected cost" parameter is the estimate of actual costs without the influence of market or organizational pressures. The "dictated cost" is the cost estimate when influenced by market or organizational pressures. For example, the dictated cost could be a proposed cost. With risk defined in this manner, a positive risk is a cost exposure, and a negative risk indicates cost protection.

For example, if your software development organization produces an estimate of 750 LM for a software project and then management decides to propose 600 LM in the cost proposal, the cost risk, (150/600)•100, is 25 percent. Obviously, by this definition, a negative result represents an actual cost risk, and a positive result represents some degree of cost surety.

Although the previous definition of risk is commonplace, this guidebook defines risk in terms of probabilities and not percentages. Perhaps the previous definition could be called "cost exposure" and not "cost risk."

6.9.2 Interval Estimates of Cost

The method in Section 6.9.1 deals with, in statistical terms, a point estimate, i.e., just one estimate of cost (750 LM) out of an implied distribution of many possible estimates. In the previous example 750 LM is, to the estimator, the most likely point or value of cost. Much better estimates are possible if you know the distribution of possible costs because then you can make an interval estimate of cost, i.e., associating a probability with a range of possible costs.

Another more desirable method is to estimate a distribution of possible costs by assigning probabilities to the range or distribution of possible sizes and unit costs of the software product. Risk then becomes defined as a probability, which is really the proper definition. The method of estimating cost risk is the equivalent of a statistical interval estimate.

To derive a distribution of possible costs, you must assign probabilities to the possible range of software product sizes in SLOC and to the possible range of labor rates in LM/KSLOC, i.e., the total LM divided by the total KSLOC. You can generate the probabilities shown in Table 6-16 using past experience as guidance, or you can generate the probabilities by surveying the software development managers and lead technical personnel to get their estimates of the probabilities and then averaging these estimates to produce a set of probabilities as in Table 6-16.

Table 6-16. Example of Size and Unit Cost Risk

Size		Unit Cost	
KSLOC	**Probability**	**LM/KSLOC**	**Probability**
100	0.10	6.500	0.15
150	0.25	5.500	0.20
200	0.50	4.500	0.40
250	0.15	3.500	0.25
Total	1.00	Total	1.00

You can estimate size using any of the techniques shown in Section 5, and you can build a range of possible values around the figures you develop. With that estimate in mind, you should develop a range of size and labor rates by varying some of the assumptions underlying the estimate until you have covered the

range of possible values of size. You should divide the range of size values into equal intervals and, using experience as a guide, assign probabilities. You can generate unit cost probabilities in the same way.

Next you cross-tabulate the unit cost values with the size values and, for each size-unit cost combination, create two values. The first value is the product of the size probability with the unit cost probability. This value indicates the probability of the occurrence of this size-unit cost pair, i.e., the probability of the associated cost. The second value is the product of size with unit cost and indicates the cost (in LM) associated with that pair. Then the two values generated for each size-unit cost combination (pair) are:

$$\text{Probability(Pair)} = \text{Probability(Size)} \cdot \text{Probability(UnitCost)}$$

$$\text{Cost(LM)} = \text{KSLOC} \cdot \text{LM/KSLOC}$$

In the case where the software system is composed of new and reused code, the size metric should be KESLOC, which you can determine using the methods in Section 4. If reused code costs 30 percent of new code, then:

$$\text{KESLOC} = 1.0(\text{KSLOC}_{new}) + 0.30(\text{KSLOC}_{reused})$$

Table 6-17 shows the cross-tabulation and generation of the distribution probabilities with the LM cost estimates. The probabilities in Table 6-17 have been multiplied by 100 to express them as percentages of (the area under) the distribution.

Table 6-17. Example of Derivation of Distribution of Costs

Probability/Unit Cost (LM/KSLOC)	Probability/Size (KSLOC)			
	0.10	0.25	0.50	0.15
	100.00	150.00	200.00	250.00
0.150	1.50	3.75	7.50	2.25
6.500	650.00	975.00	1,300.00	1,625.00
0.200	2.00	5.00	10.00	3.00
5.500	550.00	825.00	1,100.00	1,375.00
0.400	4.00	10.00	20.00	6.00
4.500	450.00	675.00	900.00	1,125.00
0.250	2.50	6.25	12.50	3.75
3.500	350.00	525.00	700.00	875.00

Now you can order the LM values with their associated probabilities, as shown in Table 6-18. You can compute risk by reference to a table of cost versus cumulative probability, such as Table 6-18. Risk is defined as the difference between the cumulative percent probability of any given LM estimate and 100 so that, if you make a decision to propose the software at a cost value of 1,100 LM, the risk, for the example shown in Table 6-18, is 18.75 percent.

Table 6-18. Example of Distribution of Costs

Cost (LM)	Probability (Percent)	Cumulative Probability
350	2.50	2.50
450	4.00	6.50
525	6.25	12.75
550	2.00	14.75
650	1.50	16.25
675	10.00	26.25
700	12.50	38.75
825	5.00	43.75
875	3.75	47.50
900	20.00	67.50
975	3.75	71.25
1,100	10.00	81.25
1,125	6.00	87.25
1,300	7.50	94.75
1,375	3.00	97.75
1,625	2.25	100.00

So now you can say that there is "only" a probability of 0.5875 that your cost estimate of 750 LM will be exceeded while there is a 0.8450 probability (i.e., an 84.50 percent chance) that the management-proposed cost value of 600 LM will be exceeded. The difference in these probabilities may cause the proposal value of cost to undergo further review. If a 20 percent risk is acceptable, then the corresponding value (interpolated from Table 6-18) of 1,084.4 LM will be the new proposed cost value of the software. Figure 6-1 shows this cost risk graphically.

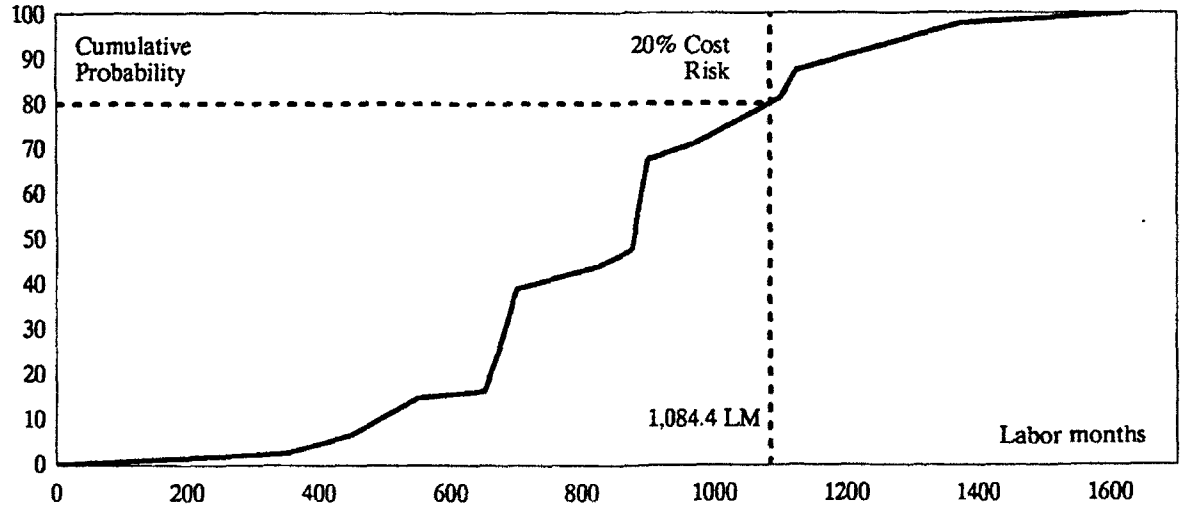


Figure 6-1. Cumulative Distribution of Costs

The calculation of risk provides management with information it can employ as a rational basis for making business decisions about what is a good, proposed price based on knowledge of the cost risk. The cost risk is one of several major factors involved in the development of a proposal price.

You should calculate cost risk for every software development project.

6.10 ORGANIZATIONAL CONSIDERATIONS

The recommended sequence of estimation activities is to first estimate the size of the software product, then estimate the cost, and finally estimate the development schedule based on the size and cost estimates. You can revise these estimates as often as available resources permit.

Your software development organization and an organization independent of your organization should make estimates of software development costs. This independent organization, which might be the measurements group or the cost engineering group, should produce size, cost, and schedule estimates free from organizational and market pressures. These cost estimates will serve as a reference point for cost estimates that are made taking organizational and market pressures into consideration, as in a proposal situation. The difference between the final accepted cost estimates and the original cost estimates will be an indication of the cost exposure. You should calculate the risk associated with each cost estimate.

This recommendation is not saying that your software development organization should not have responsibility for software cost estimation. Of course it should, and you should make all estimates in parallel. But project managers or proposal managers need to know more than just estimates of cost. They need to know cost exposure and risk. Only an independent estimating organization that is not subject to organizational and market pressures can provide this type of information.

If several organizationally independent groups produce cost estimates, they should discuss them and negotiate with the software development group. If the estimates differ by more than 10 to 20 percent and cannot be reconciled, then the issue should be taken to higher management with a statement of the risk involved. The principal point here is that making business decisions should be separate from obtaining the information on which those decisions should be based.

You should use multiple cost-estimating methods (including size-estimating methods) whenever possible because one method can serve as a useful cross-check on another. When the results do not agree to within 10 (or 20) percent, then you should reconcile them.

It is important to realize that a cost estimate is not the same as a price, which in turn is not the same as a budget. A cost estimate is an estimate of the true total costs of software development based on all the facts at hand. Management, with this estimate in hand, may well decide to take a cost risk and propose a different (usually lower) cost from which the price is computed. The proposed cost becomes the basis of negotiation with the customer, and the cost that emerges from these negotiations is actually a negotiated price including a profit. When the business is won, budgets will be established based on the negotiated price. These budgets will often be lower than those derived from the negotiated price since it is standard procedure in many organizations to withhold 10 to 15 percent of every budget as a "management reserve." Thus the negotiated price may be lower than the cost estimate, and the budget may be lower than the negotiated price.

6.11 SOFTWARE MAINTENANCE COSTS

Software maintenance consists of enhancements to the software system and the correction of errors (defects). Thus, you can view software maintenance as software development. The difference between a software maintenance situation and a new software development situation is that in maintenance, the new system (being maintained) becomes the reused code and the code developed during maintenance becomes the new code.

Section 8 will discuss the patterns of defect discovery in version 2 of this guidebook. Enhancements to software are (relatively) small development projects, usually done in response to an engineering change proposal (*ECP*). This document treats defect correction and enhancements under a single category called "problems." (Cruickshank 1988) reports that experience with a real-time operating system showed that there were 350 ECPs for a 62 KSLOC operating system, or about 6.0 problems per KSLOC during development. Other experience in software maintenance showed that problems cost about 0.6 LM to fix.

Estimates of defects per KSLOC existing at delivery, based on actual software design review and code inspection error data, show that man-rated (space) software might be expected to have about 0.02 defects/KSLOC at delivery. Ground-based software can have about 0.3 defects/KSLOC at delivery, and airborne and seaborne software can have about 0.7 to 1.0 defects/KSLOC at delivery. You can use these figures to estimate the costs of error correction, or you can derive your own. Keep in mind that, once you estimate the cost of maintenance and enhancements, the additional costs of support to software discussed in Section 6.8 also apply.

6.12 SUMMARY OF RECOMMENDATIONS

Table 6-19 summarizes the recommendations on cost estimation dependent on process maturity levels presented in Section 6.

Table 6-19. Recommendations on Methods for Estimating Software Costs

SEI Process Maturity Levels 1 and 2	SEI Process Maturity Levels 3 Through 5
Begin estimating costs with basic COCOMO, gradually introducing intermediate COCOMO.	Derive values for labor rates from experience data, and use activity-based models to estimate costs.
Use SLIM for tradeoffs among size, schedule, and effort.	Use SLIM for tradeoffs among size, schedule, and effort.
Establish an enterprise software experience database. Use the data to customize holistic model parameter values.	Use experience data to "feed back" for process improvement.
Establish formal cost estimating procedures.	Customize procedures and models to the enterprise software development environment. Develop and refine labor rates for activity-based models using experience data.
Use several holistic models to cross-check the cost and schedule estimates made by each.	Use estimates of documentation effort to cross-check activity-based estimates for document-driven activities. Use estimates of testing effort to cross-check activity-based estimates for testing activities.

The general recommendations presented in Section 6 are:

- Use LM or LH for initial computation of software costs. Using these cost units facilitates cost comparisons. You can convert the LM or LH to dollars when appropriate.
- For accurate cost estimation, have a defined and managed software development process.
- Make estimates of software cost throughout the development life cycle. The models you use to estimate cost will depend on the process maturity level of your development organization. Generally, the lower maturity levels should use holistic models, and the higher maturity levels should use activity-based models.
- Cross-check cost estimates made by holistic and activity-based models at all times. The type of cross-checking is dependent on the process maturity level.
- Customize software costing models, metrics, and parameter values to the enterprise software development environment. The type of customization is dependent on the process maturity level.
- Consider the effect of an HOL and of product size on development costs.
- Use top-down models at the earliest stages of system conceptualization to estimate the allocation of resources (effort) to the general tasks to be accomplished.
- Estimate the costs of support to software development for every project.
- Make estimates of software development costs in parallel with an organization such as measurements or cost engineering that is independent of software development. Negotiate differences where they are substantial. If differences cannot be reconciled, then inform higher management.
- Estimate cost risk and cost exposure for every project. Use the information resulting from estimates made in parallel to provide estimates of cost risk and cost exposure.

7. HOW TO ESTIMATE SCHEDULE

7.1 SCHEDULE ESTIMATION OVERVIEW

It is important to accurately schedule (i.e., to estimate the development time) of a completely new software product or of a new version of an existent software product. If the schedule is too short, the software product may not contain all of the required functions. If the schedule is too long, the product may be delivered late. It is also important to be able to perform schedule/development effort tradeoffs and to create a staffing curve for the project development labor schedule. When doing project planning and applying the methods discussed in this section, you should ask the following schedule-related questions:

- How long should I expect the development to take?
- What is the likely effect on the development effort of my shrinking or enlarging the development schedule from what has either been imposed on me or which I have estimated to be required?
- What is the staffing profile (LM) for this project?

To address these questions, this section presents methods that tell you how to:

- Estimate a development schedule, given that you know (or have an estimate of) the size of your software product and how much effort it will take to develop it.
- Make a tradeoff between between the length of the development schedule and the effort required to develop the product.
- Determine whether a schedule given to you is compatible with the size of a proposed product and the effort estimated to be required for its development.
- Develop a spread of software development labor over the development time (schedule) that you have estimated.
- Estimate the potential impact on the software development schedule of incorporating reused code into the new software product that you are developing.

When planning the development of a new software product, you can make tradeoffs among cost, schedule, and size. For example, if you want a lower cost, then you must reduce the size from the figure originally contemplated. Schedule (the period of time for software development) is a key consideration in planning for a software development project. You would expect the effect of varying quality requirements to impact schedule and/or cost. Often you can ensure higher quality software, in part through

more extensive testing. This increases the development effort over that which you would require for a software product that does not need to be of that quality level.

7.2 ESTIMATING THE DEVELOPMENT SCHEDULE

This section tells you how to estimate the length of time, t_d, required to develop a software product, given that you know (or have estimated) its size (see Section 5) and how much effort you will need to make to do so (see Section 6). To estimate t_d, use the formula:

$$t_d = \left(\frac{S}{C * K^p} \right)^{1/q}$$

where S is the software product size in SLOC (excluding comments) or ESLOC when there is reused code involved (see Sections 4 and 7.3), C is the technology constant which numerically represents both the complexity of the software to be developed and the sophistication of the development environment, K is the development effort in labor years, and t_d is the development schedule (design through installation) in years.

You can also compute the schedule for this period plus that covering the creation of requirements using this formula, but with the figures for K and C adjusted accordingly.

This equation is based on the Putnam or SLIM model discussed in Section 6.2.2 which is:

$$S = C * K^p * t_d^q$$

Here, you use the parameter values p = 0.6288 and q = 0.5555 following (Gaffney 1983). If you do not know the value of the technology constant C from previous experience, you can infer it with guidance from Table 7-1. The software modes in Table 7-1 are from (Boehm 1981) and also are defined in Section 6.2.1.

Table 7-1. Inferred Technology Constant Values

SEI Process Maturity Level	Value of SLIM Technology Constant (C) for Software Mode		
	Organic	Semidetached	Embedded
1	12,000	10,000	6,000
2	30,000	25,000	15,000
3	42,000	35,000	21,000
4	54,000	45,000	27,000
5	78,000	65,000	39,000

Now consider an application of the equation for estimating t_d. Suppose that C = 10,000, S = 300,000, K = 167.5 labor years (equivalent to a development productivity of 150 SLOC/LM), p = 0.6288, and q = 0.555. Then you estimate the estimated schedule, t_d, in years as follows:

$$t_d = \left(\frac{300,000}{10,000 * 167.5^{0.6288}} \right)^{1.8} = 1.4$$

This equation estimates only the overall development schedule. You should not use it to estimate the length of time necessary to do each of the activities or phases that compose the overall development process.

You should estimate the overall development schedule for every project. You should also make the schedule estimation and testing methods part of your software standards and policies.

7.3 SCHEDULE IMPACT OF REUSED CODE

To examine the effect of code reuse on your (estimated) development schedule, determine the size in ESLOC of a software system using the equation:

$$ESLOC = S_N + S_R\,(C_{VR}/C_{VN})$$

where C_{VN} is the unit cost (LM/KSLOC or LH/SLOC) of new code in an application system, C_{VR} is the unit cost of reused code in an application system, S_N is the amount of new code in the application system, and SR is the amount of reused code in the application system.

To relate the length of the development schedule for cases in which the system consists of all new code and in which it consists in part of reused code, let:

- K_N = Effort (labor years) to develop an application system composed of all new code.
- K_R = Effort to develop an application system consisting of both new and reused code.
- P = The relative productivity enhancement to be found in developing the system when reuse is involved as compared with the case in which it is not.
- t_{dn} = Development schedule (months or years) for an application system of size S ESLOC composed of all new code.
- t_{dr} = Development schedule for an application system of size S ESLOC composed of both new and reused code.
- R = The proportion of code reuse: $R = S_R/(S_N + S_R)$.

(Gaffney and Durek 1988) give the following relative schedule reduction for reusing code instead of writing all new code:

$$\frac{t_{dr}}{t_{dn}} = P^{\frac{(p-1)}{q}}$$

where:

$$\frac{K_R}{K_N} = \frac{1}{P} = \frac{C_{VN} * (1-R) + C_{VR} * R}{C_{VN}} = 1 + R * \left(\frac{C_{VR}}{C_{VN}} - 1\right)$$

You may use the relation for K_R/K_N for various parametric what-if analyses to estimate the possible effect on the development schedule of various amounts of code reuse.

As an example, let $C_{VN} = 5.000$ LM/KSLOC, $C_{VR} = 0.375$ LM/KSLOC, and $R = 0.90$. Then $1/P = 0.1675$, and $P = 5.97$. Using the values of p and q given in Section 7.2 (0.6288 and 0.555):

$$\frac{t_{dr}}{t_{dn}} = P^{\frac{(p-1)}{q}} = P^{-0.6682} = 0.30$$

The schedule to develop the software product containing new and reused code would be only 30 percent as long as that to develop the same product with all new code.

7.4 SCHEDULE/DEVELOPMENT EFFORT TRADEOFF

Suppose that you have used the Putnam (SLIM) equation of Section 7.2 to estimate that the "ideal" length of the schedule for your software product is t_0 for your product, based on an estimate of its size (see Section 5) as S ESLOC and of development labor effort of K_0 labor years with a technology constant of C. Now suppose that you want to consider the effect on the amount of labor years required due to changing the schedule to t_1 (a figure which has been imposed on you). The labor years will become K_1 which is calculated using the equation:

$$K_1 = K_0 \cdot \left(\frac{t_0}{t_1}\right)^{q/p} = K_0 \cdot \left(\frac{t_0}{t_1}\right)^{0.8834}$$

This equation is the schedule/development effort tradeoff equation and is derived from the Putnam equation given above.

As an example, suppose there was a 20 percent schedule reduction. Then $t_0/t_1 = 1/.8 = 1.25$. Suppose the originally estimated effort was $K_0 = 50$ labor years. Then the effort for the case in which the schedule was reduced by 20 percent would be $K_1 = 60.9$ labor years or an increase of 22 percent. This calculation illustrates the effect of schedule compression that you can often expect on development effort.

You should make schedule/development effort tradeoff studies for all software development projects and products. Tradeoff methods should be part of your enterprise's software standards.

7.5 SCHEDULE/EFFORT/SIZE COMPATIBILITY

You should determine whether the figures (estimates or objectives) for schedule, size, and effort for your project are compatible. The customer may impose the length of time for development, or you may perceive a need to get a new product out into the marketplace quickly. You can determine whether the figures are mutually compatible by estimating the value of the technology constant C (defined in Sections 6.2.2 and 7.2) implied from figures for schedule t_d, effort K, and size S that you have been given or otherwise calculated. You can calculate C from the equation:

$$C = \left(\frac{S}{t_d^{0.5555} \cdot K^{0.6288}}\right)$$

On the basis of the value determined for C, you can determine whether a given schedule is compatible with the size and effort proposed for the project by using the process shown in Figure 7-1.

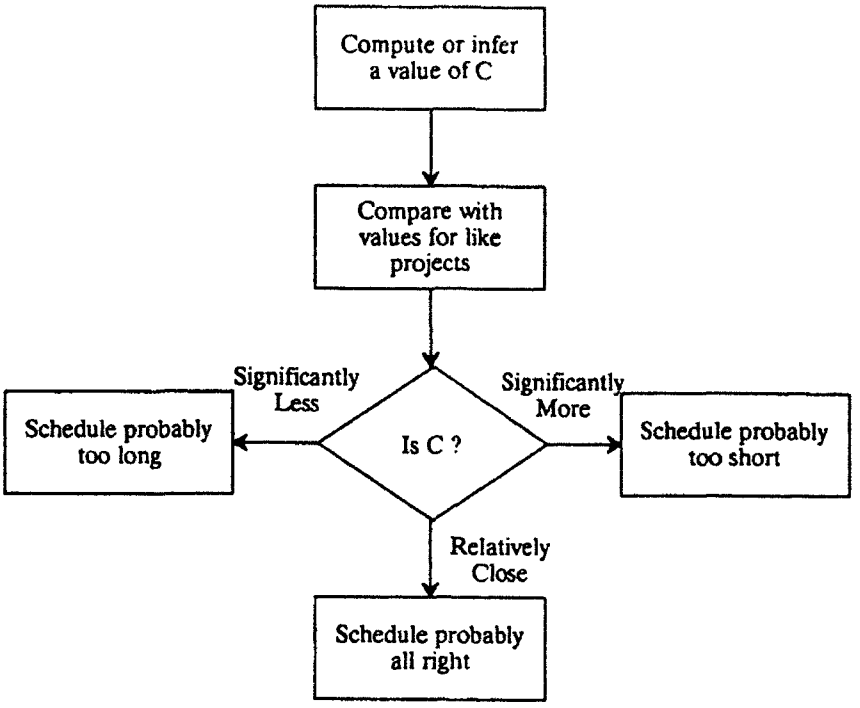


Figure 7-1. Schedule Compatibility Testing Process

7.6 SOFTWARE DEVELOPMENT LABOR PROFILES

This section tells you how to estimate the spread of development labor over the schedule of length t_d months. Unless otherwise noted in this section, the unit of t_d is assumed to be months rather than years, as was done in the preceding section. The labor profile, or spread, is the number of LM over the schedule period. The total figure of K labor months is to be spread over t_d months.

You should think about your project's activities in detail when doing labor resource spreading. This section shows you how to do an overall spread without consideration of the individual activities that constitute the development project you are planning. Because the method uses only figures for K and t_d, you may need to do some adjusting.

The most common form of labor spreading is based on the Rayleigh distribution. The "instantaneous" or density form of the Rayleigh distribution is:

$$y(t) = \frac{E}{t_p^2} \cdot t \cdot e^{-\frac{t^2}{2t_p^2}}$$

The cumulative form is:

$$Y(t) = E \cdot \left(1 - e^{-\frac{t^2}{2t_p^2}}\right)$$

where t is the time, Y(t), E is the total area under the curve to infinity, and Y(t) is the area under the curve to time t.

Thus, $Y(t_d) = K$ would be the area under the curve, or the total labor expended through the period of development, t_d. This is the K total development labor used in the previous sections. As described below, let $K = 0.999\ E$ when applying the Rayleigh distribution to the estimation of the time profile of development labor application. The parameter t_p is the location of the peak of the instantaneous curve. Figure 7-2 shows the form of this instantaneous density curve over the development life cycle.

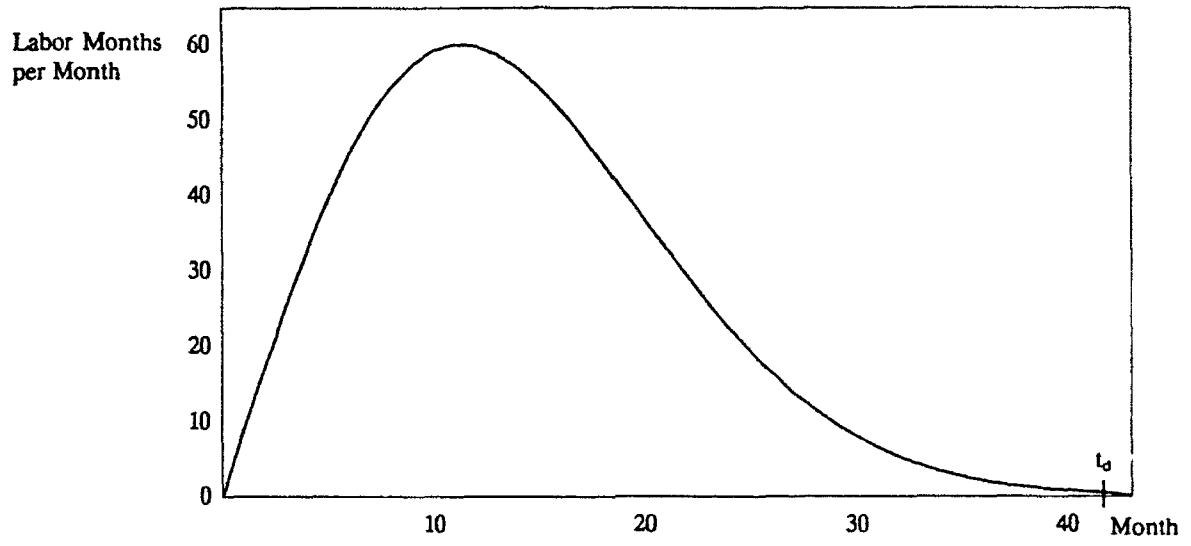


Figure 7-2. Development Effort Planning Curve

This is a "practical" form of the Rayleigh curve that you can use when planning the labor allocations for a software development project. This form recognizes that:

- Actual development effort does not continue to infinity as does the (continuous) Rayleigh curve.
- Actual project effort is applied in discrete intervals of time (taken to be one month in length here).

Using the discrete form, the effort, E_i, for interval i (LM if you have given your development period, t_d, in months) is given by the equation:

$$E_i = Y(i) - Y(i-1) = \frac{K}{0.999} \cdot \left(e^{-\frac{(i-1)^2}{2t_p^2}} - e^{-\frac{i^2}{2t_p^2}} \right)$$

where K is the total development effort equal to $K(t_d)$ as explained above. Note that the 0.999 in the formula makes the area under the "instantaneous" or labor density curve from $t = 0$ to $t = t_d$ equal to K, and t_p is the location of the peak of the period of development. You can compute this from t_d, the length of the schedule, by the relation $t_p = t_d/3.7169$. Note that i is the interval number and that there are $N = t_d/12$ one-month intervals in a development period of t_d months. The 0.999 in the formula makes the area under the "instantaneous" or labor density curve from $t = 0$ to $t = t_d$ equal to K.

For example, suppose that K = 1,116.0 LM and the schedule, t_d, is 42 months. Then the area under the Rayleigh curve to infinity, E, will be 1,116/0.999 = 1,117.7 LM. Using the cumulative form of the Rayleigh curve, you have:

$$0.999 = 1 - e^{-\frac{42^2}{2t_p^2}}$$

Therefore:

$$-\frac{42^2}{2t_p^2} = \ln(0.001) = -6.9078$$

and therefore $t_p = 11.30$ months.

As a check:

$$E_{42} = 1,117.1 * \left(1 - e^{-0.003916*(42^2)}\right) = 1,116.0.$$

which is the value of the effort stated above. So for any month (e.g., month 2), the estimated LM is:

$$E_2 = 1,117.1 * \left(e^{-0.003916*(1^2)} - e^{-0.003916*(2^2)}\right) = 13.0$$

You should generate a labor profile or staffing curve for every software development project.

7.7 SUMMARY OF RECOMMENDATIONS

The recommendations on schedule estimation presented in Section 7 are:

- Estimate the schedule for every project and product development, and test this estimated schedule for compatibility with the size and the estimated development effort of the intended product.
- Conduct schedule/development effort tradeoff studies for every project and product.
- Make schedule estimation methods, compatibility testing methods, and tradeoff methods part of your enterprise's software standards.
- Generate a labor profile for every software development project.

This page intentionally left blank.

8. ESTIMATING DEFECT CONTENT

This page intentionally left blank.

9. MANAGEMENT INDICATORS FOR TRACKING AND MONITORING

9.1 MANAGEMENT BY MEASUREMENT

This section presents quantitative methods and procedures for tracking the product, process, and progress of a software development project. Project management personnel routinely determine progress, compare actuals to projections, analyze reasons for discrepancies, and develop corrective action if needed. These management actions require a continuing process of measurement. The establishment of quantitative objectives, the monitoring of project status, and the evaluation of product and process quality all involve measurement of some kind. This section discusses how to measure the process, product, and progress aspects of a software development project and how to use these measurements to make judgments about that project on which you can base actions such as making improvements to the process.

9.1.1 Software Development Project Tracking and Monitoring

To manage a software development project, you must first collect project measurement data and then form metrics out of the measurements. Section 4 defines a measurement as a number assigned to a primitive. You may functionally combine one or more measurements to form a metric. The metrics discussed in this section are called "management indicators" because they indicate the status of progress, process, or product and because they should help you make judgments about the project. Management does not need to understand the technical details of data collection or the construction of the indicator metrics since the measurement function (however organized) performs the data collection and analysis. Your organization should define the data collection methods, indicator construction, and software development planning and surveillance procedures in its software development standards and policies. Management, however, should understand what the indicators are revealing about the project, and management should be able to take the appropriate actions to solve any problems indicated by the tracking indicators.

The process of managing a software development project through the use of these indicators is called "tracking" or "monitoring," and this section uses these terms interchangeably. The tracking process consists of periodically collecting data, calculating indicator metrics from that data, and organizing reports which indicate the status of the project. The measurement data often indicates the status, such as the total LM spent to date. The rules to follow in using tracking metrics are:

- Establish the metrics to be used and the measurements to be collected in advance of the start of the effort.
- Relate the metrics to action that you can take to rectify problems (or at least limit the damage).

- Collect measurements during the project. Do not reconstruct them after the fact.
- Collect an appropriate set of measurements to aid you in setting realistic goals for the future, to enable you to determine the degree of improvement, and to provide input to your estimating algorithm.

You should track large software development projects using the methods presented in this section. The definition of a large project should be in your enterprise's software development standards, but some organizations have tracked only those projects above 100 LM of effort (for example).

9.1.2 Project Surveillance and Process Maturity Level

One of the SEI requirements (2.3.1) for the achievement of process maturity level 4 is that you establish a managed and controlled process database for process metrics data. You should collect such data across all projects. Section 4 of this guidebook discusses the enterprise software experience database in terms of not only process data but product data (size, cost, schedule, etc.) that you should collect upon project completion. The SEI process maturity scheme suggests that you can feed back this data to the process to improve the process. This guidebook suggests that you can feed back this data to improve the product, i.e., future software products.

There is, however, another aspect to data collection. Section 4 treats the collection of "actuals" at the completion of the project for historical analysis and both process and product improvement. You must collect data **during** the project for process and product improvement while in development. You should also collect data during development to obtain an accurate indication of project status. You can keep this data in the enterprise software experience database or in a separate database. Section 9 discusses the collection and analysis of this "in process" data and how you can use it for tracking and monitoring.

You should begin data collection and analysis for both historical and project surveillance purposes while your enterprise is at process maturity level 1 or 2. There is no need to wait until level 3 to do so. The organization at level 1 or 2 should be able to feed back the experience data for incremental process and product improvement, whether historical or immediate. The quantitative surveillance of a software development project by tracking progress, process, and product during development through data collection, analysis, and management action is characteristic of a level 4 or 5 organization.

9.2 MANAGEMENT INDICATORS

Many sets of software management indicators have been published, forming a list of hundreds of metrics. Obviously, you cannot perform effective tracking with hundreds of indicators, so you must select a manageable set at the initiation of the project. Extensive discussions of indicators can be found in (Grady and Caswell 1987; Schultz 1988; Air Force Systems Command 1986; Army Materiel Command 1987; and National Aeronautics and Space Administration 1990). This guidebook presents 38 metrics which you can use to track a software development project. Many of these indicator metrics have been taken from these references. Many of them have been redefined to be more precise. The product completion indicator (*PCI*) is an entirely new metric, not presented in any other reference. The earned value metric indicates status in terms of equivalent (to new) source statements complete, thus allowing a comparison with the total source statements to be developed and giving a precise indication of the proportion of software complete.

It is important to recognize that an indicator can have several metrics associated with it. For example, you can measure the cost-to-date indicator in LM, LH, or dollars. If your enterprise software standards do not specify cost units, you, with the help of the measurements group, must decide how to measure the cost-to-date indicator. You might select either LM or dollars or both. If your enterprise dictates no standard set of software development indicators in its software standards and policies, you must make these choices before the project begins.

Table 9-1 presents the management indicators grouped by measurement category. You should view this table as a "menu" from which you can select the necessary indicators for tracking according to the needs of the given project. Effective tracking depends on selecting a set of indicators small enough to be manageable and economically viable, and large enough to contain all of the information necessary to manage efficiently.

Table 9-1 indicates a subset of 20 of the 38 indicators labeled as the "minimum data set." This minimum set indicates the smallest possible set to gain the minimum information to manage the product development and the process feedback in an efficient manner. A single asterisk designates these indicators.

You should use the data collected during the development process for tracking product, progress status, and process improvement. You should feed back the collected data to incrementally optimize the software development process. In addition, you should use at least the minimum data set for tracking software development projects. Your enterprise should define its own data set for tracking.

Table 9-1 also indicates the three indicators that are of special importance in the development of embedded systems. A double asterisk designates them.

Table 9-1. Software Management Indicators and Metrics

Number	Measurement Category	Indicator	Metrics	Minimum Data Set	Embedded System
1	Size	Current estimate or count	New, reused, and total KSLOC (or function points)	*	
2			KESLOC		
3		Percent current estimate is of estimate at initiation	(Current/initial)100	*	
4	Cost	Cost to date	LM		
5			LH	*	
6			Dollars ($)	*	
7		Percent of budget spent to date	Percent LM		
8			Percent LH	*	
9			Percent $	*	
10	Schedule	Elapsed development time	Elapsed months		
11		Percent of schedule elapsed	(Elapsed months/ schedule months)100	*	
12	Stability	ECPs	Count ECPs	*	
13		Percent requirements undefined	(Requirements to be defined/ total requirements)100	*	
14		Number of software action items (*SAIs*)	Count SAIs		

Table 9-1, continued

Number	Measurement Category	Indicator	Metrics	Minimum Data Set	Embedded System
15		Percent SAIs closed to date	(SAIs closed/total SAIs)100		
16		Authorized positions staffed	Count people		
17		Percent planned positions staffed to date	(Staffed/planned)100	*	
18	Status	Percent requirements designed	(Requirements designed/ total requirements)100		
19		Percent requirements coded	(Requirements coded/ total requirements)100		
20		Percent requirements tested	(Requirements tested/ total requirements)100		
21		Percent tests passed	(Tests passed/total tests)100		
22		Percent measurement units (KSLOC, function points, CSUs, or CSCs) designed to date	(Units designed/total units)100	*	
23		Percent measurement units (KSLOC, function points, CSUs, or CSCs) coded (including CSU test) to date	(Units coded/total units)100	*	
24	Status	Percent measurement units (KSLOC, function points, CSUs, or CSCs) tested (including CSC test) to date	(Units tested/total units)100	*	
25	Status	Percent measurement units (KSLOC, function points, CSUs, CSCs, or CSCIs) integrated (including CSCI test)	(Units integrated/total units)100	*	
26	Quality	Number of defects per KSLOC in preliminary design reviews	Defects or errors in preliminary design reviews/KSLOC (use actual or estimated KSLOC)	*	
27		Number of defects per KSLOC in detailed design reviews	Defects or errors in detailed design reviews/KSLOC (use actual or estimated KSLOC)	*	
28		Number of defects per KSLOC in code inspections	Defects or errors in code inspections/KSLOC (use actual or estimated KSLOC)	*	
29		Number of (valid) PTRs opened	Count (valid) PTRs	*	
30		Percent of PTRs closed to date	(PTRs closed/total PTRs)100	*	
31		PTRs per KSLOC in CSC test	PTRs/KSLOC		
32		PTRs per KSLOC in CSCI test	PTRs/KSLOC		
33		PTRs per KSLOC in system test	PTRs/KSLOC		
34		Predicted defects/KSLOC at delivery	Use SWEEP method	*	
35	Product completion	Overall proportion of software (in KSLOC, function points, etc.) complete	See Section 9.3.7		

Table 9-1, continued

Number	Measurement Category	Indicator	Metrics	Minimum Data Set	Embedded System
36	Computer resources	Target *CPU* processing speed (for standard functions)	(Target *mips*/host mips) x (function size in mips/host processing second) = estimated target mips for standard function		**
37		Proportion of memory utilization (words, bytes, characters, or bits)	CPU used/CPU available or mass storage used/mass storage available		**
38		Proportion of software I/O capacity utilized	(Message length)(arrival rate)/ (processing speed)		**

You monitor a software development process by using information derived from each and every activity that composes the process. Detailed or precise monitoring is by the individual activities, and higher level (less detailed) monitoring is by phase of development, where a phase is a subset of activities. For example, the coding phase consists of the individual activities of coding, code inspections, and CSU testing; the detailed design phase consists of the individual activities of detailed design, design reviews, and CDRs. The project indicator calculation example in Section 9.3.7 is stated in terms of phases rather than activities. Either method should be satisfactory.

You can track the indicators in Table 9-1 and present them graphically as well as numerically. You can also track the actual measurements or metrics instead of the percentages if so desired.

9.3 HOW TO COMPUTE MANAGEMENT INDICATORS

9.3.1 Software Product Size Indicators

You should track the size of a software system using indicator 1 or indicators 2 and 3 because the delivered size is almost always larger than the initial estimate. Also, there are often memory or CPU processor constraints that restrict program size, and these restrictions can become critical in the later stages of development. You should monitor the size of the software product (i.e., the new and the total [new plus reused] KSLOC) through the full development schedule. You can also monitor function points if you use them. If estimates of the software product size are available, then you should track the estimated size. If actual code counts are available, you should track the actual code counts. While the software product is in design, you can still track the KSLOC by converting the counts of source lines of design or process bubbles to estimates of KSLOC. The past experience of the enterprise design process, as recorded in the enterprise software experience database, will reveal the SLOC-to-SLOD ratio. (Some specific experience shows a ratio of about 4, but each enterprise must calculate its own ratio.) You can also track function points continuously throughout the development process.

9.3.2 Software Cost Indicators

Indicators 4 through 9 relate to development cost. You should monitor dollars and either LH or LM, and you will monitor them relative to the budget. You should track internal software development costs for staff in terms of LH or LM but track expenses such as computer support, consultants, subcontractors, and equipment in terms of dollars. Also track total dollars spent to date. When placed in the enterprise software experience database, annotate this data with the date on which they were incurred so that you can relate or compare them with similar experiences on other projects at different times.

9.3.3 Development Schedule Indicators

Schedule tracking is standard practice on every software development project. Schedule tracking, as in indicator 10, is usually in terms of months. The percent of schedule elapsed, as in indicator 11, is often a help in comparing time resources spent (for example) to financial resources spent. For example, you should want to compare percent of schedule elapsed with percent of budget spent, and if the percent of budget spent is significantly different from the percent of schedule elapsed, then you must take action.

9.3.4 Project Technical Stability Indicators

Project technical stability is the extent to which the requirements of the software development are undefined from the time of initiation of the project. This lack of definition is manifested principally in incomplete and contradictory technical requirements and in insufficient human resources. Insufficient schedule and budgets are also related to technical instability, but these situations are covered by other management indicators.

Technical stability of the software development project, as shown by indicators 12 through 17, is important since stability is often the prime determinant of whether or not the project will be completed within the required cost, schedule, and quality requirements imposed upon it. The volume of ECPs and the percent of undefined requirements are key indicators that you should track. It is often difficult to find a sufficient number of qualified personnel, and a shortage of such people can severely impact the schedule. You should closely monitor the number of unfilled software development positions.

9.3.5 Project Status Indicators

You can monitor the status (i.e., the degree of project completion) by measuring the degree of implementation of requirements through the major phases of development, as shown by indicators 18 through 20. To achieve a greater degree of precision, however, you should express the project status in terms of the percentage of measurement units that have been completed at each of the major phases of development, as shown by indicators 21 through 25. Measurement units can be SLOC, function points, CSUs, CSCs, or even CSCIs. The unit commonly selected is SLOC, but you can use any unit. Since the actual SLOC size is not known at the design phases, you may use SLOD or estimated SLOC. You can also convert from SLOD to SLOC by using a standard SLOC-to-SLOD ratio derived from experience.

9.3.6 Quality Indicators

The data records you normally keep as part of design reviews and code inspections should provide the data for the quality indicators 26 through 28. You can use these indicators, together with quality indicators 31 through 33, as input to methodologies and tools such as the SWEEP method (Gaffney and Pietrolewicz 1990) to predict software defects per KSLOC at delivery. Since the actual SLOC size is not known at the design phases, you may use SLOD or estimated SLOC. You can also convert from SLOD to SLOC by using a standard SLOC-to-SLOD ratio derived from experience.

9.3.7 The Product Completion Indicator

To calculate the PCI, indicator 35, weight the number of source statements (KSLOC) that have completed each activity or phase by the effort (e.g., LM/KSLOC) required for each activity or phase.

Then divide the total of the weighted SLOC by the total effort (LM/KSLOC) for software development to produce an overall proportion complete (*OPC*) for the software product. You can multiply this proportion by the total SLOC to be developed to give the number of equivalent (to new) source statements completed. This quantity is the indication of product completion. You can compare this number with the (estimate of) the total SLOC to be produced in the software product to show status.

Section 9.2 discussed the distinction between the activities of software development and the phases of software development. Basically, phases are groups of individual activities that compose the development process. For example, detailed design is the phase of development consisting of the individual activities of detailed design, internal design reviews, and CDRs. The activity-based tables of Section 6 show both the phases and the activities that compose the phases. The method of calculating the PCI shown in this section allows you to organize the calculation by either phase or activity. The example shown employs a phase-based method, and you should use this method. The method presented here is a variation of the earned value calculation method.

The steps in computing the PCI are:

- Characterize each activity or phase in the development process by a labor rate in LM/KSLOC or equivalent units. The enterprise at SEI process maturity level 3 or 4 should know its development process(es) at least that well.
- Determine the proportion of software, measured in SLOC or other designated measurement units, that is through each phase or activity. You can obtain status data from the software status report (if any) and from interviews with the software development manager.
- Compute the OPC software by multiplying the labor rate by the proportion complete for each activity or phase. Compute the adjustment for work in process for each activity or phase. Divide by the total labor rate. Do this calculation separately for each CSC or CSCI since each is likely to have a unique status in terms of proportion complete through the activities. Perform this calculation for the development process used in the project or, if you use more than one development process, for each unique process.
- Multiply the OPC for each CSC or CSCI by the corresponding (estimate of) SLOC (or other unit) to obtain the PCI or earned value in terms of the designated measurement units such as SLOC. Add the PCIs over all of the CSCs or CSCIs to obtain the total equivalent product completed in SLOC or other designated measurement units.
- If desired, compute the equivalent (to new) product completed productivity to date by dividing the total equivalent product completed by the LM expended to date.

The PCI is a very valuable metric that should always be tracked and yet seldom is because of a lack of knowledge about how to use the applicable metrics. A measurement of product completion is not the same as a measurement of labor expended to date. The PCI is a measure of the useful work that has actually been accomplished to date in terms of a standard unit of measurement (such as equivalent SLOC completed). In comparison, labor expended to date is just the expenditure of LM or LH, perhaps with very little useful work having been accomplished. The expenditure of LH or dollars does not necessarily mean that any given useful work in terms of product has been accomplished.

The development process of a computer program is composed of phases or activities 1 through n, with corresponding unit costs of C_1 though C_n and with a total unit cost of $\Sigma C_i = T$. Unit cost here is in

LM/KSLOC and size is in KSLOC. You can also use LH/SLOC and SLOC. Let the proportion of total units through each phase or activity be $P_1, P_2, \ldots, P_n$, where $P_1 \geq P_2 \geq \ldots \geq P_n$. Then the OPC of the software is:

$$OPC = \left[\sum_{i=1}^{n} C_i P_i + \sum_{i=1}^{n} C_i\left(\frac{P_{i-1} - P_i}{2}\right)\right] \cdot \left(\frac{1}{T}\right) = \left[\sum_{i=1}^{n} \frac{C_i(P_i + P_{i-1})}{2}\right] \cdot \left(\frac{1}{T}\right)$$

The PCI results from multiplying the OPC by the number of units (e.g., source statements) under development. The example that follows is organized in terms of phases. The PCI indicates how many equivalent KSLOC have been completed, i.e., the earned value.

$$PCI = (OPC)(\text{Total KSLOC})$$

This model of product completion assumes that not only are P_i of the units through activity i, but that P_i of the units are, on the average, halfway through activity $i + 1$. This later assumption allows for the inclusion of work in process in any activity. If you assume that work in process is only one-third through activity $i + 1$ on the average, then you can change the divisor from 2 to 3.

As an example of the use of the PCI metric, suppose that a CSCI has a development process and present status as described in Table 9-2. Further assume that 87 percent of the units (KSLOC) are under development (i.e., $P_0 = 0.87$, or 87 percent of the CSCI is somewhere in the process), but 60 percent of the CSCI is through preliminary and detailed design and 45 percent of the CSCI is through code and unit test.

Table 9-2. Example of Product Completion Indicator Calculation

i	Activity/Phase Complete	C_i (LM/KSLOCt)	P_i (Proportion of KSLOC of Final Software Product Complete)
0	In process (started)		0.87
1	Preliminary design	0.52	0.60
2	Detailed design	0.82	0.60
3	Code and CSU test	2.21	0.45
4	CSC integration test	1.48	0.10
5	CSCI test	1.22	0.00
	Total (T)	6.25	

Using the previous formula, the OPC is:

$$OPC = [0.52(0.60) + 0.82(0.60) + 2.21(0.45) + 0.74(0.10) + 1.22(0.0)$$

$$+ 0.52(0.135) + 0.82(0.0) + 2.21(0.075) + 0.74(0.175) + 1.22(0.05)](1/6.25) = 0.37$$

The OPC is 0.37, and if there are 140 KSLOC in the CSCI, then there are the equivalent of $(0.37)(140) = 51.8$ KSLOC complete, and 51.8 KSLOC is the equivalent product completed to date.

The KSLOC in the example refers to estimates of the software product size. In the design phases, KSLOC does not exist, but you can generate estimates of the equivalent KSLOC designed by converting

the design units to KSLOC, e.g., multiplying the thousand source lines of design (*KSLOD*) by a standard SLOC-to-SLOD ratio or multiplying the number of process bubbles designed by a standard KSLOC-to-bubble ratio. You can derive such conversion ratios from previous experience.

Productivity is not among the management indicator metrics discussed here because productivity is very difficult to evaluate while the software product is in development. Productivity is seldom tracked in a development project. The observed productivities change as the product moves through the various phases and activities of development, and there are few standards by which to evaluate these observed productivities. If you divide indicator 1 by indicator 4 to produce an "observed SLOC/LM to date" metric, there is no way to judge whether this number is acceptable or not. You can only evaluate productivities and compare them for completed (by whatever process) source statements or equivalent to completed source statements and for processes that are composed of the same phases or activities.

The PCI metric, however, is an "equivalent to completed development" metric. It is a KSLOC estimate that is the weighted equivalent to a completed software product. Therefore, you can divide the PCI (times 1,000 to convert it to SLOC) by the cost to date to obtain an equivalent to new code productivity, and you can compare this productivity with any standard or experienced new code productivities that exist. You can use this method to track productivity. In the preceding example of product completed, suppose that 400 LM is the cost to date (indicator 4). Then the equivalent to completed (new) code productivity to date is 51,800/400 = 130 SLOC/LM. At any point in the development process, if the PCI is strictly a function of the LM expended, the productivity will remain constant.

9.3.8 Computer Resources Indicators

Sometimes you track computer resources, as shown by indicators 36 through 38, especially when you have doubts about the existence of sufficient computer resources. Sometimes the software specification requires that the software to be developed or a specific software function must execute at a minimum specified speed on a CPU that is specified but not yet developed or otherwise unavailable. To approximate and monitor target processing speed (as shown by indicator 36) when the target CPU is unavailable, you must first identify the key software functions that have the potential to impose major constraints on the speed of system processing. These functions may be frequently used routines or modules that have (perhaps) some difficulty in handling the information flow. Examples include fast Fourier transforms, Kalman filtering, message handling, or I/O routines.

For any of these functions, the ratio of the specified target CPU speed in mips to the host (development) CPU speed in mips represents the ratio of processing speeds for these particular functions or for the developmental software in general, whatever the specification requires. If you divide the software function size in millions of instructions by the number of seconds it takes the host CPU to process the software function (or multiple iterations of the function), you obtain the software function host execution speed in mips. You can multiply this host execution speed by the ratio of processing speeds to yield an estimate of the target CPU software processing speed in mips.

You can monitor this CPU processing speed indicator as you develop the software function or functions. The early software versions may indicate a target processing under the specification, but as you develop and optimize the software with respect to speed (among other factors), the estimated target process speed should approach the specification.

Indicator 37, memory utilization, is calculated by dividing the CPU utilization by the CPU available capacity. You can measure memory in words, bytes, characters, bits, or instructions. You can calculate

the proportion of mass storage utilization by dividing mass storage utilization by mass storage available. In addition, you can use tracks and cylinders as viable measures of memory for disk units.

Indicator 38, I/O facility utilization, requires knowledge of the statistical distribution of message length, where messages are the units of information that are processed by the I/O software. With knowledge of this distribution, you should select a high message length of about three standard deviations above the mean. You should convert this message length, which you express in characters, words, or bytes, to the number of bits per message. Multiply this length in bits by the arrival rate of the messages (in or out) in messages per second. Then divide this product by the processing capacity in bits per second of the I/O software that prepares and sends messages to or receives messages from the hardware channel. The result of these calculations is the proportion of I/O software capacity used. You can monitor this proportion (i.e., you can calculate and compare it with the specification) as you develop and optimize the software with respect to message handling capacity.

9.4 DATA FOR SOFTWARE DEVELOPMENT PROJECT SURVEILLANCE

9.4.1 Data Sources

An organization developing complex systems has many sources of information that it can use for software development tracking. Table 9-3 shows the indicator groups, the organization that should control the data, and the (possible) data sources or documents that should contain the data.

Table 9-3. Data Sources for Software Management Indicators

Measurement Category	Indicator Group	Organization	Data Sources
Size	Current estimate or count of size	Software development (library) or configuration management	Software development plan; count from programming environment or CASE tool; configuration management reports; software library
Cost	Cost	Finance	Accounting reports
Schedule	Elapsed time	Software development	Software development plan
Stability	ECPs	Systems engineering (change control) or project management	ECP status report; program management reports
Stability	Undefined requirements, satisfied requirements	Systems engineering (change control)	Requirements analysis reports; requirements traceability matrix
Stability	SAIs	Software development	Software status report
Stability	Project staffing	Software development	Software status report
Status	Software progress	Software development	Software status report
Quality	Current and predicted defects (reviews and inspections)	Software quality engineering (assurance)	Software quality plan; software quality status report
Quality	PTRs	Software quality engineering or system test	PTR status report
Project completion	PCI	Software development	Software status report
Computer resources	Computer resources	Software development or systems engineering	Software and system development plans

9.4.2 Surveillance Activities

The implementation of tracking and monitoring procedures is really a continuous process of measuring the product and process, comparing those measurements and metrics with the goals and limits set by the project plans, and taking corrective action when the performance falls outside the preset limits or fall short of the preset goals. Figure 9-1 illustrates this monitoring and control process as a flowchart.

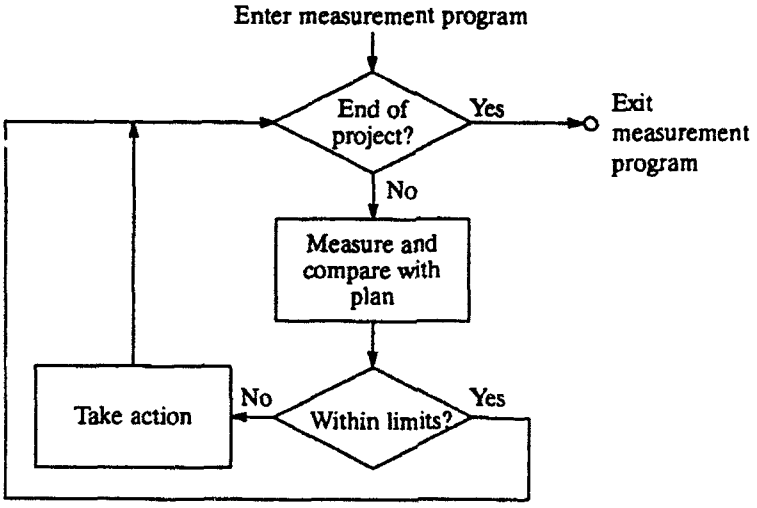


Figure 9-1. The Monitoring and Control Process

9.4.3 Data Collection

Tables 9-4 and 9-5 illustrate some measurements data collection forms that you can use to organize and calculate management indicators and status for a software development project. You should collect data while the project is in progress. For large projects, it is better to collect measurements by CSCI. For smaller projects, you can collect data directly for the CSCs themselves. The data collection forms implied by Tables 9-4 and 9-5 allow for both types of data collection, and they are organized around the minimum data set presented in Section 9.2.

The top part of Table 9-4 contains the CSCI identification, the language used (if there are two or more languages in the same CSCI, use the corresponding number of forms), the date of data collection, the size in counted (or estimated) SLOC (or function points), the number of CSUs, the number of prototypes or iterations if you use the spiral model, and the estimate of size made at the initiation of the project.

Fill out the center part of Table 9-4 with the same sort of status and process data that is shown in the example in Table 9-2. Use the data to calculate project completion, just as in Section 9.3.7.

Fill out the lower part of Table 9-4 for the CSCI (or CSC if desired) with the basic cost data, both budget and spent to date. Use this data to compute an estimate at completion (*EAC*) to compare with the budget and to predict overruns or underruns.

Table 9-4. Management Indicators Work Measurements Collection Form

CSCI/CSC			Language			Date	
SLOC (Function Points)		CSUs		Prototypes		Initial Estimate	
Development Process—Indicate Units of Measurement (SLOC, Function Points, etc.)							
Phase/Activity	% Effort (Total=100)			Phase/Activity		% Complete	
				Started			
Preliminary design				Preliminary design			
Detailed design				Detailed design			
Code and CSU test				Code and CSU test			
CSC integration and test				CSC integration and test			
CSCI test				CSCI test			
Expected Overall Productivity in Units/LM for Above Process							

CSCI/CSC	Phase/Activity	Cost Account	Budget		Spent to Date	
			LM/LH	$	LM/LH	$
Host System				Target System		

Table 9-5. Management Indicators Quality Measurements Collection Form

Quality/Reliability Measurements—Number of Reviews/Inspections					
	Defects			PTRs	
CSCI/CSC	Preliminary Design	Detailed Design	Code Inspections	CSC Integration and Test	CSCI Test

Number of Engineering Change Proposals			
Number of requirements		Requirements defined	
Number of planned positions		Number of positions staffed	
Project software start date		Months elapsed	
Planned software completion date			

9.5 THE ESTIMATE AT COMPLETION

The EAC is an estimate of the completed cost of the software product, i.e., a prediction of the actual cost. At the initiation of development, the EAC is equal to the estimated actual cost of the software product. However, during development, the EAC is composed of the inception to date (*ITD*) cost (the actual cost to date, also known as cost to date), and the estimate to complete (*ETC*) (the current estimate of the actual additional cost to complete the remaining development of the software product). You can state this relationship in equation form as:

$$EAC = ITD + ETC$$

Section 9.3.7 showed how to calculate the OPC, the product completion in terms of equivalent (to new) KSLOC complete, and the completed equivalent to new code productivity. The equivalent to new code productivity was the PCI divided by the cost to date. You can state this quantity as PCI/ITD. In labor rate terms, this quantity is ITD/PCI, which is measured in LM/KSLOC for equivalent to completed new code development.

In Section 9.3.7 the PCI was:

$$PCI = (OPC)(\text{Total KSLOC})$$

and therefore the number of equivalent KSLOC yet to be developed is (1 - OPC)(Total KSLOC).

It seems reasonable to use the actual labor rate to date (in terms of equivalent to completed new code labor rate) as the projected labor rate to complete the software development. Therefore, compute the ETC as:

$$ETC = [(1 - OPC)(KSLOC)]\left[\frac{ITD}{PCI}\right]$$

Using the PCI (earned value) example from Section 9.3.7:

$$ETC = [(1 - 0.37)(140)]\left[\frac{400}{51.8}\right] = 681$$

Thus the ETC is 600 LM and the EAC is:

$$EAC = ITD + ETC + 400 + 681 = 1{,}081 \text{ LM}$$

Compare this estimate with the initial cost estimate which is (using the total labor rate from Table 9-2) (6.25)(140) = 875 LM.

9.6 GRAPHICAL METHODS OF MONITORING AND CONTROL

Figures 9-2 through 9-7 show graphical methods of project monitoring. (Air Force Systems Command 1986) shows some alternative graphical methods. Figure 9-2 shows a graphical method of monitoring the total defects (errors) per KSLOC of a software development project. The defects per KSLOC metric evolves to a stable level, and if this level is within limits, you do not need to take any action. If this level is above the maximum acceptable limit, then perhaps the unexpectedly high defects per KSLOC is caused by the possibility that the software development process is introducing too many errors. If the defects per KSLOC are below the minimum limit, then perhaps you are not applying the error discovery process according to the standard. You can also graphically track status as shown in Figure 9-3.

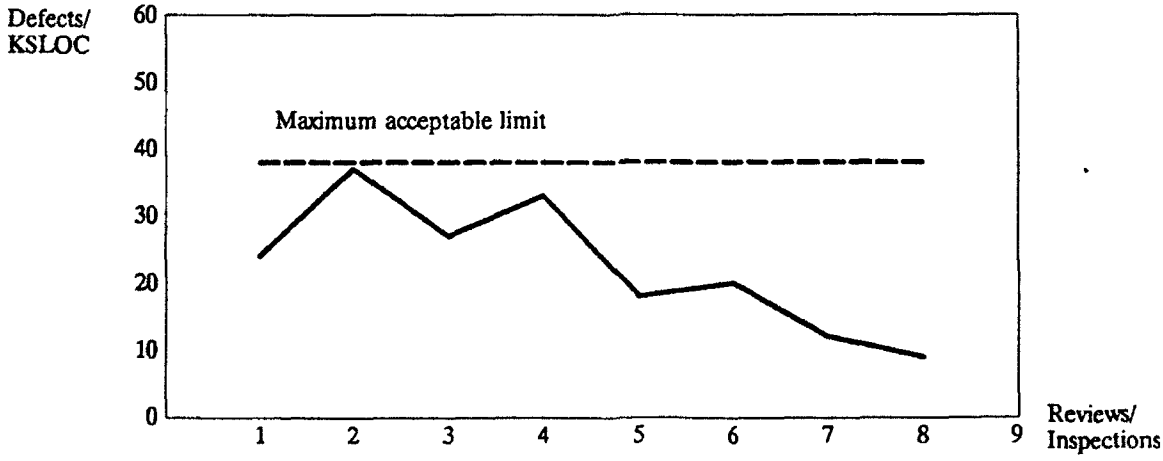


Figure 9-2. Example of Monitoring Defects Per Thousand Source Lines of Code by Review or Inspection

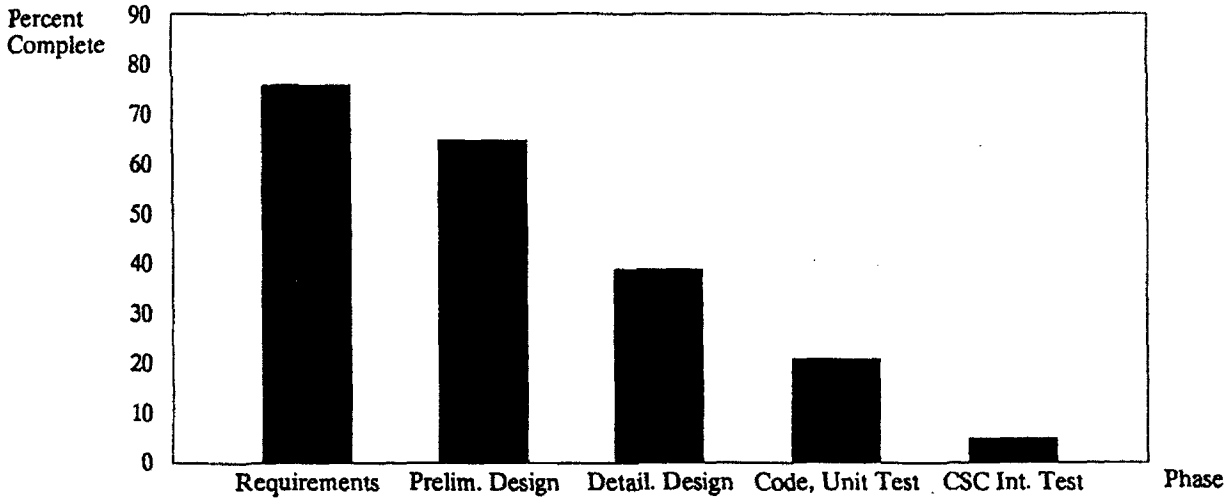


Figure 9-3. Example of Status Tracking

Figures 9-4 and 9-5 show patterns of PTR opening and closure. A software development project in control should experience an upward trend of PTR opening with increasing testing. When the problems begin to be solved, the trend should turn downward. If the trend is continually upward with more and more PTRs being opened, a very large project-level problem exists. If PTR closures fall off in the presence of increasing or steady PTR openings, there is a problem. Perhaps a continuing closure effort is not being made, or perhaps the problems to be solved have become much more difficult. However, if PTR closures fall off in the presence of declining PTR openings, then the closure procedure is just catching up with the volume of PTRs, and this condition is normal.

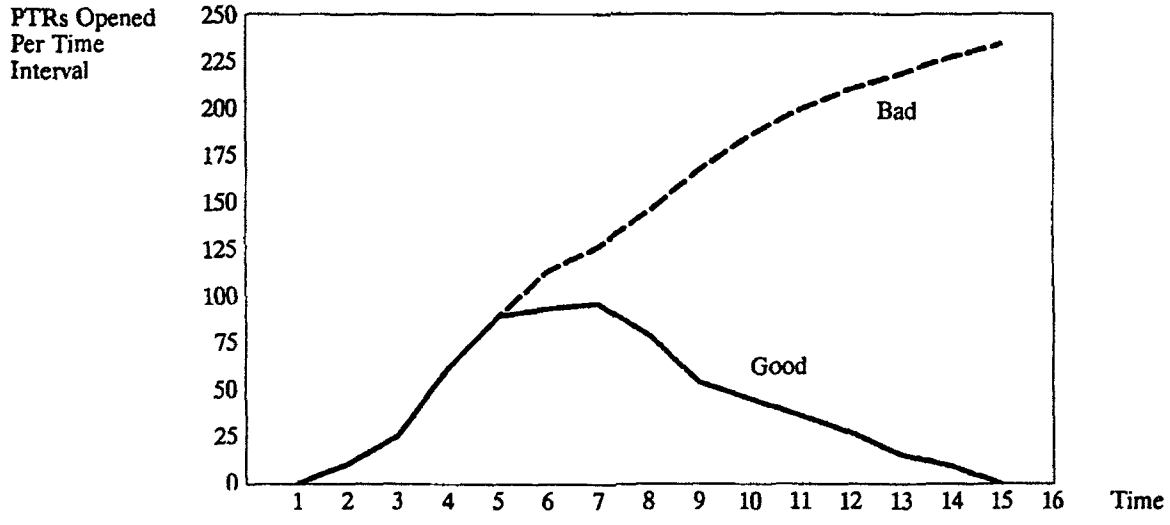


Figure 9-4. Patterns of Program Trouble Report Opening

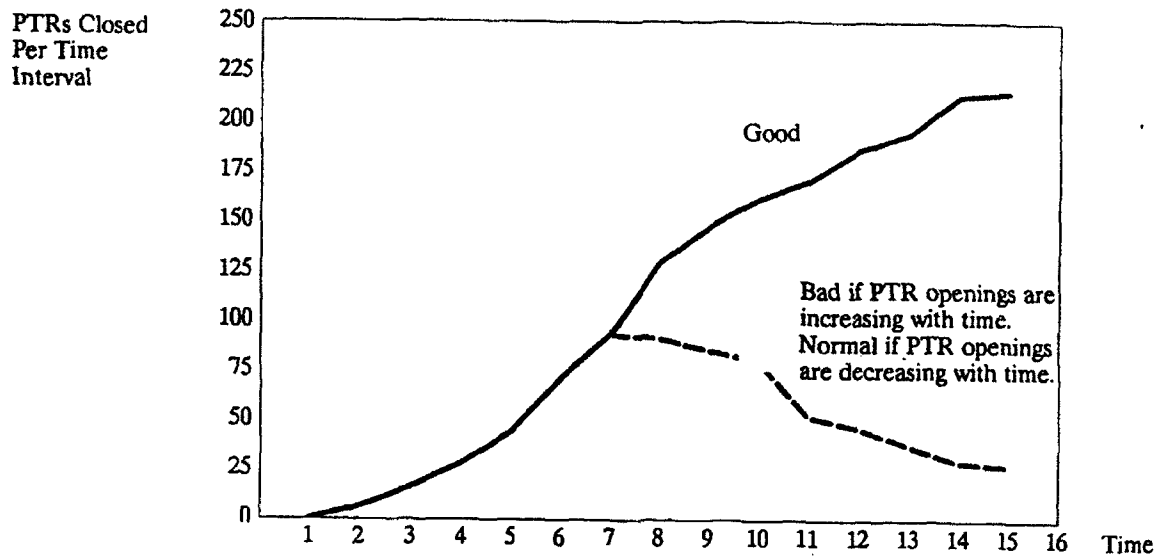


Figure 9-5. Patterns of Program Trouble Report Closure

Figure 9-6 illustrates how you can monitor defect density with preset control limits. The limits can be statistical control limits based on previous projects in the enterprise as well as goals for the new project. When defect discovery rates (i.e., defects/KSLOC) go outside the control limits, you must closely examine the development process. Figure 9-7 shows a similar monitoring example for a computer resources indicator.

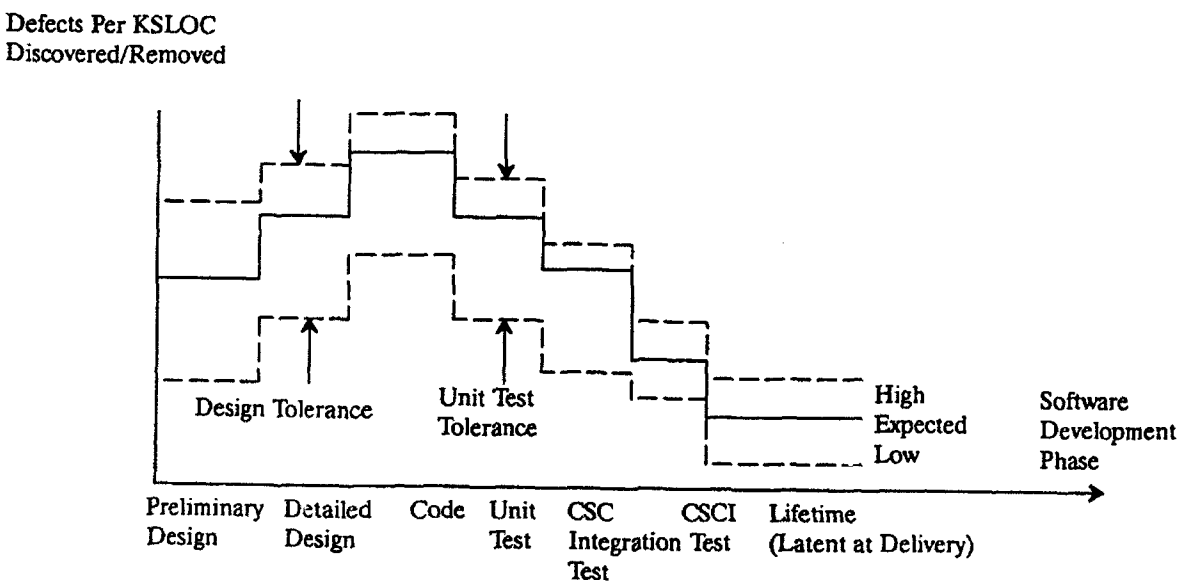


Figure 9-6. Example of Defect Density Monitoring and Control

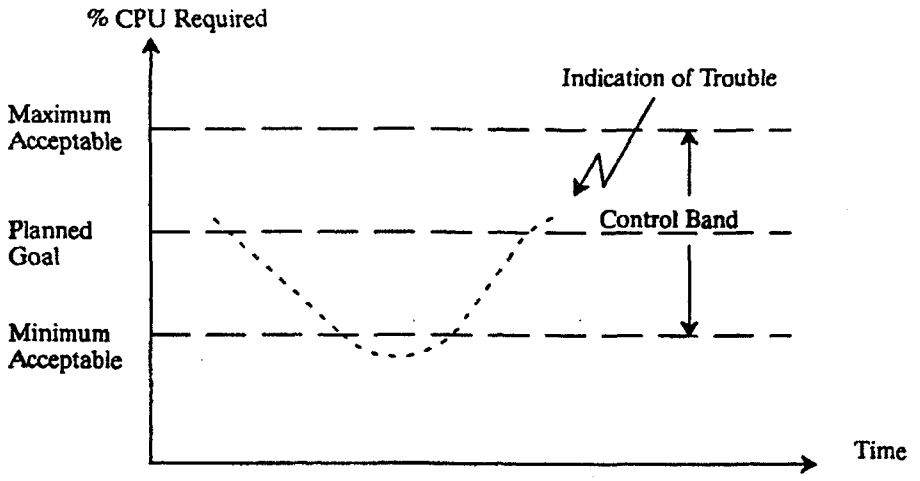


Figure 9-7. Example of Computer Resource Monitoring and Control

9.7 SUMMARY OF RECOMMENDATIONS

The recommendations on project tracking and monitoring presented in Section 9 are:

- Define data collection methods, indicator construction, metrics definition, and project surveillance procedures in your enterprise software standards and policies.
- Track large software development projects. Your enterprise should define procedures for deciding what projects are to be tracked.
- Have your enterprise define its own tracking data set and its own management indicators.
- Use at least the minimum data set for software project tracking.
- Begin data collection for tracking at the earliest possible time, regardless of process maturity level. Feed back data to improve the process and the products.

This page intentionally left blank.

10. PROCESS MATURITY HIERARCHY: FRAMEWORK FOR MEASUREMENT

This section describes the measurement technology requirements for attaining higher SEI software process maturity levels. Successful attainment of level 2 simultaneously builds the foundation for future progress to levels 3, 4, and 5. The section shows how this guidebook answers the **measurement-related** SEI assessment questions associated with SEI process maturity levels 2 through 5. It also shows how implementing the quantitative management methods described in the preceding sections can help produce higher quality and more usable software products and processes.

10.1 OVERVIEW

Figure 10-1 shows that an effective measurement activity, which is **necessary** for level 2, is the essential underpinning for all activities needed to manage software projects at SEI levels 3 and above. It is literally impossible for an organization to progress to higher levels of SEI process maturity until it firmly establishes the measurement program.

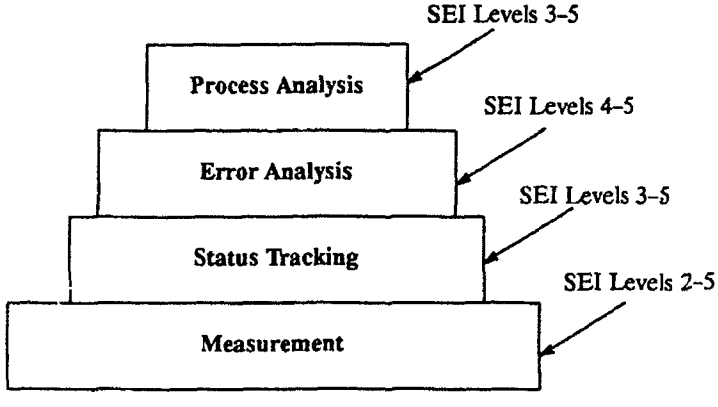


Figure 10-1. Measurement Foundation for Software Engineering Institute Levels Three Through Five

Table 10-1 synthesizes key requirements given in (Humphrey, Kitson, and Kasse 1989; and Humphrey 1989). The table characterizes the five levels of SEI process maturity and shows the actions required to reach the next higher level. The fourth column highlights the typical performance found in the 1988 SEI survey of more than 100 assessments completed.

Table 10-1. Levels of Software Process Maturity

Process Level	Typical Characteristics	Required Actions to Reach Next Level	SEI Survey Results—1988
1. Initial	***Chaotic.*** Professionals are driven from crisis to crisis by unplanned priorities and unmanaged change. Surprises cause unpredictable cost, schedule, and quality performance.	Need: Process must become repeatable. Requires measurement and planning (of size, cost estimates, and schedules), performance tracking, change control, better management of project commitments, and quality assurance.	**86 percent at level 1.** Of these, 66 percent do not estimate software size, 64 percent do not track size changes, 58 percent do not track errors in code and test, 49 percent do not have formal procedures to schedule and estimate, and 45 percent do not review software commitments.
2. Repeatable	***Intuitive.*** Reasonable control of schedules; highly variable cost and quality; informal and ad hoc process methods and procedures.	Need: Process must become defined. Requires developing process standards and definitions, assigning process resources, and establishing methods for requirements, design, inspection, and test.	**13 percent at level 2.** Of these, 88 percent do not have formal software engineering training, 77 percent do not have formal regression testing, 50 percent do not collect design error data, 31 percent do not have a formal software process group, and 31 percent do not have formal quality assurance.
3. Defined	***Qualitative.*** Reliable costs and schedules; improving but still unpredictable quality performance,	Need: Management must focus on process, not product. Requires establishing process measurements and quantitative quality goals, plans, measurements, and tracking.	**Only 1 percent at Level 3.**
4. Managed	***Quantitative.*** Reasonable, statistical control over costs, schedules, and product quality.	Need: Process must become optimizing. Requires quantitative productivity plans and tracking, instrumented process environment, and economically justified technology investments.	**None at Levels 4 or 5.**
5. Optimizing	***Optimizing.*** Quantitative basis for continuous process improvement; continued capital investment in process improvement and automation.	State: Continuous process improvement. Requires continued emphasis on process measurement and process methods for error prevention.	**None at Levels 4 or 5.**

10.2 PROCESS MATURITY LEVELS

In seven of every eight software development projects considered, the SEI assessments showed the state of software development practices corresponding to process maturity level 1, the chaotic initial level. SEI found that, in level 1 software development organizations, software professionals are driven

from crisis to crisis by unplanned priorities and unmanaged change. The inevitable surprises cause chaos, leading to unpredictable cost, schedule, and quality performance.

10.2.1 Process Maturity Level One

The level 1 environment is labeled "chaotic" because it employs ill-defined procedures and controls. To overcome that operational chaos and progress to level 2's "repeatable" process, enterprises must institutionalize rigorous management of project commitments and priorities, costs, schedules, quality assurance, and control of the changes that occur as projects evolve. Implementing a planned measurement program is the first action. Until they know what they are doing, their occasional software successes will not be repeatable.

Level 1 organizations need to develop the ability to repeat their successes in developing software and to overcome the serious schedule and cost problems they face. That is, they need to **consistently** apply software engineering management to their software development process and use modern tools and technology. An effective measurement ability is the foundation for increasing a software development enterprise's process maturity level. Only two-thirds typically use formal methods to estimate the size of their software products and to track changes in product size during development. Similarly, only half formally schedule and estimate their costs or formalize project team commitments in advance.

The third column of Table 10-1 shows that level 1 organizations need to concentrate on:

- Measurement and planning (of size, cost estimates, and schedules).
- Performance tracking.
- Change control.
- Better management of project commitments.
- Quality assurance.

To develop their ability to consistently apply a software engineering capability, as shown in Table 10-1, level 1 organizations begin by routinely collecting measurements on software size, schedules, and development cost.

10.2.2 Process Maturity Level Two

An enterprise at process maturity level 2 has developed a repeatable process. It estimates and controls its project schedules, although cost and quality may continue to be highly variable, and it still may use many informal and ad hoc methods and procedures. In developing software, the level 2 enterprise uses standard methods and practices for cost estimating, scheduling, requirements changes, code changes, and status reviews. To track project performance and commitments and to baseline their software development process steps and quality assurance activities, level 2 organizations routinely maintain profiles over time for:

- Staffing.
- Units designed, build/release content, units completing test, and units integrated.

- Test progress.
- Utilization of memory, throughput, and I/O channels.
- Errors found during inspections of designs and during software code and test.

Striving to join the best one percent, at level 3, involves defining a software development process. The level 2 organization establishes definitions, methods, and standards for its processes of requirements, design, inspection, and test. This improves its assignment and control of process resources and increases the predictability of its process results.

10.2.3 Process Maturity Level Three

Level 3, "defined process" organizations collect the data described for level 2 enterprises. To define their methods for processes (requirements, design, inspection, and test) and to better assign resources to process steps, they must:

- Maintain formal records for unit development progress.
- Maintain formal records for test coverage.

In moving toward process level 4, the level 3 organization develops a managed and controlled process database for process metrics across all projects and updates it routinely with current measurement and metric data.

10.2.4 Process Maturity Level Four

"Managed processes" typify level 4 organizations. Building on the measurement foundations established in levels 2 and 3, they expand their focus from individual software projects to their software development process and to measuring and improving specific process steps. In addition to monitoring each software development project, level 4 organizations:

- Maintain a managed and controlled database for process metrics across all projects, a database that is updated routinely with current measurement and metric data.
- Project, compare actual performance to quality goals, and analyze errors found during design inspections and in code and test reviews.
- Routinely analyze software productivity for major process steps.

10.2.5 Process Maturity Level Five

Continuing process improvement is institutionalized in organizations at level 5. They routinely feed back process measurements and results to further improve the software development process.

10.2.6 Summary of Measurement-Related Activities

Table 10-2 summarizes measurement-related activities in levels 2 through 5. Level 3 functions concentrate on defining and institutionalizing the organization's software development process. Level 4 focuses on a

quantified management process, and Level 5, the level of continuing process improvement, focuses on the "optimizing" process by incorporating lessons learned from results of process modifications and development experience.

Table 10-2. Measurement-Related Activities by Process Maturity Level

Level 2, Repeatable Process	Level 3, Defined	Levels 4 and 5
Estimate, plan, and measure software size, resource estimates, staffing levels, schedules, and development cost. Maintain profiles over time for units designed, build/release content, units completing test, units integrated, and test progress, requirements changes, and staffing. Maintain profiles over time for utilization of target system memory, throughput, and I/O channels. Collect statistics on design errors and on software code and test errors found in reviews and inspections.	Level 2 data, plus: • Maintain formal records for progress of unit development. • Maintain formal records for test coverage.	Levels 2 and 3 data, plus: • Routinely project, compare to actuals, and analyze errors found in reviews and inspections of requirements, designs, and code and test. • Routinely analyze software productivity for major process steps. • Maintain managed and controlled process database for process metrics across all projects.

10.3 QUANTITATIVE MANAGEMENT FUNCTIONS BY PROCESS MATURITY LEVEL

Measurement-related functions provide the necessary foundation upon which organizations raise their capabilities to higher process maturity levels. However, measurement-related activities by themselves are not sufficient to raise an organization's process maturity level. Management attention and action to ongoing organizational activities is also required for:

- Training, with standard, **required** courses for specific project tasks (e.g., inspections, reviews).
- Rigorous management of key activities.
- Basic defined, documented, and followed software development methods and standards.

Table 10-3 summarizes the most important management activities by process maturity level. The portion describing standard courses provided by the training function differentiates subject matter by process maturity level. In particular, note that organizations with repeatable processes offer training in planning, estimating, and tracking for various software product attributes, change control, configuration management, and the organization's internal processes for managing commitments. When these organizations reach level 3 and have defined processes, they will also provide training in quality management, inspections, basic software development methods, and management of software professionals. Training in level 4 and level 5 organizations also include quality planning, quantitative process management, advanced development methods such as prototyping, and career planning for software professionals.

Table 10-3. Management Activities by Process Maturity Level

Level 2—"Repeatable" Process	Level 3—"Defined" (Customizable "standard" process)	Levels 4 and 5 (Measured, analyzed process)
Standard Courses Provided by Training Function:		
Planning, estimating, and tracking product size, checkpoint performance, resource requirements, and staffing levels Change control and configuration management Organization's process of commitment/approval/ accountability Subcontract management	Quality management Managing software professionals Basic software methods Inspections Courses required for each job function which are defined in training plans	Quality planning Quantitative process management Advanced development methods Prototyping Planning and development of technical careers
Organization Rigorously Manages:		
Commitments Requirements, changes, and size of software products Schedules Cost	Project performance, which is tracked and reviewed by a key activity in the defined standard process Design and code inspections Measurements of errors found and costs incurred by process activity Development and documentation of process standards and methods Development tools, methods, process definitions, and standards which are under CM for each project	Process database Project performance against quality plan for each project Product defect levels, inspection and test coverage and efficiency, error distribution, task productivity, and effectiveness of tools and methods Process metric definitions maintained under CM control Subcontractor standard quality metrics; tracking and reviewing subcontractor quality performance
Basic Methods and Standards are Defined, Documented, and Followed for:		
Software design, code, and test; estimating software size; projecting, planning, and scheduling resources Making changes to requirements, designs, and code Conducting reviews, inspections, and audits	Estimating resources for each key activity in defined software process Managing risks: plans identify technical and business exposures and define the process means to address them Staffing plans which address needs for special skills and knowledge of application domains	Inspections, tools, methods, quality plans, and quality tracking by process task Customization of process and environment Prototyping and quantitative design

The progression for definition of methods and standards follows a similar path. Level 2 organizations have defined methods for software design, code, and test; for estimating software size; for projecting, planning, and scheduling resources; for making changes to requirements, designs, and code; and for

conducting reviews, inspections, and audits. Level 3 methods provide for more detail, for example, in estimating resources for each key activity in the defined software process and in managing risk for technical, schedule, and cost issues. Level 4 organizations have defined methods and standards for key project tasks (inspections, tools, methods, quality plans, and tracking), for customizing the process and environment as required by specific product requirements, and for prototyping and quantitative design.

10.4 MEASUREMENT ACTIVITIES IN PROCESS MATURITY HIERARCHY

Table 10-3 shows a graphic presentation of the process characteristics. For each characteristic shown, it indicates the specific level at which it is required.

10.4.1 Software Engineering Institute Assessment Focus on Process Characteristics

Table 10-4 shows the measurement-related characteristics that represent "standard organizational practice" for process maturity levels 2 through 5. The successive process maturity levels are depicted, building on the measurement foundation at the bottom. An SEI "assessment" addresses each characteristic with at least one question, followed by detailed investigation to verify that the characteristic is indeed typical of the organization's standard processes. For an organization to be certified as having attained the next level of software process maturity, investigation of responses to SEI assessment questions must show that more than 80 percent of the indicated characteristics are present at each level of process maturity, although the organization need not satisfy all characteristics.

Table 10-4 shows highlights of the requirements for each level of process maturity, which was synthesized from (Humphrey and Sweet 1987, Addendum B, 23–27). The term "mechanism," used frequently in the table, allows for the many different ways in which different organizations can implement a characteristic. The glossary in (Humphrey and Sweet 1987, 39) defines a mechanism as:

> A process that is typically conducted by a working group of software engineering professionals who developed the code in question. It is an objective assessment of each error, its potential cause, and the steps to be taken to prevent it. While placing blame is to be avoided, such questions as mistakes, adequacy of education and training, proper (software engineering) tools capability, and support effectiveness are appropriate areas for analysis.

For clarity, the many process characteristics are grouped into four functions in Table 10-4:

- ***Measurement.*** All these characteristics must be in place at level 2. Initially concentrated on characteristics of products, metrics evolve to encompass process characteristics as well.

- ***Tracking.*** Beginning with the measurement foundation, tracking functions maintain formal records for unit development progress and ensure traceability between requirements and code at level 3. Level 4 begins to focus on the tracking process. Formal records are maintained for test coverage, and review efficiency is routinely analyzed for each project.

Table 10-4. Measurement Foundations of Software Engineering Institute Process Maturity Level Hierarchy

	SEI Process Maturity Level			
Measurement Function	**2**	**3**	**4**	**5**
Process Analysis				
Has a mechanism to identify and replace obsolete technologies.				X
Has a managed and controlled database in place for process metrics across all projects. Analyzes software productivity for major process steps.			X	X
Routinely analyzes error data from code reviews and tests to determine process-related causes of errors.			X	X
Routinely analyzes causes of errors to determine process changes required to prevent errors. Mechanism initiates actions to prevent errors.			X	X
Uses formal mechanism to assess existing designs and code for reuse. Routinely compares and evaluates technologies used in-house relative to those available externally.		X	X	X
Error Analysis				
Routinely projects, compares to actuals, and analyzes design errors and code and test errors.			X	X
Maintains formal records for test coverage.			X	X
Analyzes review efficiency for each project.			X	X
Tracking				
Maintains formal records for unit development progress.		X	X	X
Has mechanisms to ensure traceability between requirements and top-level design, between top-level design and detailed design, and between detailed design and code.		X	X	X
Routinely gathers statistics on design errors. Tracks to closure those action items resulting from design reviews and code reviews. Conducts formal test case reviews.		X	X	X
Measurement-Related				
Has formal procedures to estimate and plan software size, schedules, and development cost. First-line managers commit to their estimates of schedules and cost.	X	X	X	X
Uses a formal procedure in the management review of each software development prior to making contractual commitments to ensure periodic management review of the status of each software development project.	X	X	X	X
Routinely gathers statistics on software code and test errors. Tracks to closure software trouble reports resulting from testing.	X	X	X	X
Maintains profiles over time for units designed, build/release content, units completing test, units integrated, utilization of target computer memory, throughput, and I/O channels; test progress; and staffing.	X	X	X	X

- ***Error Analysis.*** Beginning at level 4, this function routinely projects, compares to actuals, and analyzes design errors and code and test errors. Useful for development of individual software products, routine error analysis is the foundation that must be prepared before the process analysis function can exist.

- ***Process Analysis.*** Systematic process change begins at level 3, as organizations check for potential reuse of their existing designs and code. At level 4, the second major change occurs: a managed and controlled process database is in place for process metrics across all projects. At level 5, the function institutionalizes continuing process improvement.

10.4.2 Software Engineering Institute Assessment Questions by Process Characteristics

Table 10-4 shows a graphic representation of the process characteristics. The table synthesizes key requirements in (Humphrey 1989; Humphrey and Sweet 1987).

10.4.2.1 Measurement

Table 10-4 shows that an organization qualifying for process maturity level 2 has formal procedures in place to estimate and plan software size, schedules, and development cost. First-line managers commit to their estimates of schedules and cost. These organizations use formal procedures in the management review of each software development prior to making contractual commitments and to ensure periodic management review of the status of each software development project. Finally, the organization maintains profiles over time for units designed; build/release content; units completing test; units integrated; utilization of memory, throughput, and channels; test progress; and staffing. Once in place, measurement characteristics change little for levels 3 through 5, although the measurement database improves for each level. Section 9 of this guidebook tells you what measurements need to be tracked and what other tracking metrics need to be developed. Sections 4 and 6 describe specific methods for making these measurements.

10.4.2.2 Tracking

Beginning at level 3, the organization routinely maintains formal records for unit development progress and maintains mechanisms that ensure traceability between requirements and top-level design and between top-level design and detailed design. Level 3 also requires routine gathering of statistics on software code and test errors. Organizations at this level track to closure those software trouble reports resulting from testing. Level 4 requires an organization to maintain formal records for test coverage and to analyze review efficiency for each project. For specific tracking methods, see Section 9.

10.4.2.3 Error Analysis

There are no specific requirements for systematic error analysis at levels 2 or 3. Reaching level 4 requires routinely projecting, comparing to actuals, and analyzing errors found in code and test and design inspections. However, level 2 and 3 organizations should begin collecting and analyzing software error data to reach level 4. Error analysis methods are given in Section 8.

10.4.2.4 Process Analysis

Specific requirements for process analysis functions begin at level 3, where a mechanism must be in place for assessing existing designs and code for reuse. The organization must routinely compare and

evaluate technologies used relative to those available externally. These organizations routinely analyze software productivity for major process steps, error data from code reviews and tests to determine process-related causes of errors, and causes of errors to determine process changes needed to prevent errors. Finally, level 4 organizations use mechanisms to initiate actions to prevent errors. At level 5, organizations use mechanisms to identify and replace obsolete technologies. Process analysis requires repeatable estimating, tracking, and error analysis functions.

In the process analysis function, a dramatic difference in organizational focus occurs at level 4. A managed and controlled process database must be in place for process metrics across all projects. At level 4 an organization can routinely analyze software defect data (from code reviews and tests) to determine process-related causes of errors and thus improve software productivity for major process steps. With the process-related causes of errors known, the organization can routinely analyze them to determine process changes required to **prevent** process-related errors. At level 4, a defined mechanism exists to initiate actions to prevent errors. With this structure in place, the attainment of level 5 requires only that a mechanism exists to identify and replace obsolete technologies.

10.4.3 Organizational Characteristics by Process Maturity Level

Table 10-5, which is fully as important as Table 10-4 for assessments, describes characteristics of organizational structure and policies for software development at process maturity levels 2 through 5. Table entries synthesize requirements in (Humphrey and Sweet 1987, Addendum B, 23-27; Humphrey 1989).

10.4.3.1 Level Two

Four subgroups of characteristics focus on project organization, understanding of the original and evolving requirements, periodic reviews of project progress, and soundness of the reviews:

- **Six characteristics** describe the organization for projects that involve development of software:
  - Each project involving software development has a designated software manager who reports directly to the project manager.
  - The designated software quality assurance (*SQA*) reporting channel is separate from the software development project management's channel.
  - A separate software configuration control function exists for each project that involves software development.
  - The project has a mechanism for controlling changes to code.
  - The project has a mechanism to ensure that any software subcontractors follow a disciplined software development process.
  - There is a training program for all newly appointed development managers, designed to familiarize them with management of software projects.
- **Three characteristics** identify procedural mechanisms that are used for:
  - Ensuring that software design teams understand each software requirement.
  - Controlling changes to the software requirements.
  - Proving regular technical interchanges with the customer.

Table 10-5. Organizational Characteristics of Software Engineering Institute Process Maturity Level Hierarchy

	SEI Process Maturity Level		
Measurement Function	**2**	**3**	**4/5**
Process Maturity Levels 4 and 5			
Mechanisms are used for deciding when to insert new technology into the development process, managing and supporting introduction of new technologies, and periodically assessing the process and implementing indicated improvements.			X
Process Maturity Level 3			
There is a Software Engineering Process Group function.		X	X
Required software engineering training programs are given to first-line supervisors of software engineering developers and software developers. A formal training program is required for design and code review leaders.		X	X
Software engineering is represented on a system design team. A mechanism helps identify and resolve systems engineering issues that affect software.		X	X
A designated individual/team is responsible for control of software interfaces.		X	X
Each software developer has a private, computer-supported workstation or terminal.		X	X
A standardized, documented software development process is used on each project. Process documentation describes the use of tools and techniques. Standards are applied to the content of software development files/folders, preparation of unit test cases, man/machine interfaces, and code maintainability. A mechanism ensures compliance with software engineering standards.		X	X
Mechanisms are used for verifying that samples examined by SQA are truly representative of the work performed; for configuration management of the software tools used in the development process; and for independently calling integration and test issues to the attention of the project manager.		X	X
Process Maturity Level 2			
Mechanisms are used for maintaining awareness of the state of the art in software engineering technology; for ensuring that software design teams understand each software requirement; for controlling changes to the software requirements; for regular technical interchanges with the customer; and for ensuring that any software subcontractors follow a disciplined software development process.	X	X	X
Senior management has a mechanism for regular review of the status (quality, cost, schedule) of software development projects.	X	X	X
There is a required training program for all newly appointed development managers designed to familiarize them with management of software projects.	X	X	X
Coding standards and independent audits are part of the organization's software development process. There is a mechanism for ensuring that regression testing is performed routinely.	X	X	X
Each project involving software development has a designated software manager who reports directly to the project manager. The SQA reporting channel is separate from the software development project management's channel. There is a separate software configuration control function for each project that involves software development. A mechanism is used for controlling changes to code.	X	X	X

- **Two characteristics** assure regular, periodic review of project progress:
  - Senior management has a mechanism for regular review of the status (quality, cost, schedule) of software development projects.
  - Coding standards and independent audits are part of the organization's software development process.
- **Two characteristics** ensure that the periodic reviews are soundly based:
  - There is a mechanism for ensuring that regression testing is performed routinely.
  - There is a mechanism to maintain awareness of the state of the art in software engineering technology.

10.4.3.2 Levels Three and Higher

Characteristics focus on similar issues but are increasingly concerned with conditions that produce higher quality and lower cost through reduced rework.

Figure 10-2 (page 10-19) provides a graphical cause-and-effect "fishbone" representation of the characteristics that lead to level 2 status. Figures 10-3 to 10-5 (pages 10-20 to 10-22) are similar depictions leading to levels 3, 4, and 5.

10.5 MEASUREMENT SUPPORT

Table 10-6 correlates the measurement-related SEI assessment questions (Humphrey and Sweet 1987, 23-28) with the management indicators and related material in this guidebook in Section 9, Table 9-1. The table is keyed to the SEI assessment questions. In assessments, 90 percent of the asterisked SEI questions required for each specified level of process maturity are required to be answered "yes." Nonasterisked questions require only 80 percent "yes" answers.

Table 10-6. Software Management Indicators and Metrics

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.1.4	2	9				Formal procedure ensures periodic management review
2.1.7	2	9				Independent audits for each step of the software development process
2.1.14*	2	5	Size	Current estimate or count	1	New, reused, and total KSLOC (or function points)

Table 10-6, continued

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.1.15*	2	7	Schedule	Elapsed development time	10	Elapsed months
2.1.16*	2	6	Cost	Cost to date	4	LM
				Cost to date	5	LH
				Percent budget spent to date	6	Dollars ($)
				Percent budget spent to date	7	Percent LM
				Percent budget spent to date	8	Percent LH
2.1.16*	2	9	Earned value	Overall proportion of software (in KSLOC, function points, etc.) complete	35	See Section 9.3.7
2.2.1	2	6	Cost	Cost to date	4	LM
				Cost to date	5	LH
				Percent budget spent to date	6	Dollars ($)
				Percent budget spent to date	7	Percent LM
				Percent budget spent to date	8	Percent LH
		9	Stability	Authorized positions staffed	16	Count people
			Stability	Percent planned positions staffed to date	17	(Staffed/planned)100
2.2.2*	2	5	Size	Current estimate or count	1	New, reused, and total KSLOC (or function points)
				Current estimate or count	2	KESLOC
				Percent current estimate is at the original estimate	3	(Current/initial)100
2.2.3*	3	8, 9	Quality	Number of defects per KSLOC in preliminary design reviews	26	Defects or errors in preliminary design reviews/KSLOC (actual or estimated KSLOC)
				Number of defects per KSLOC in detailed design reviews	27	Defects or errors in detailed design reviews/KSLOC (actual or estimated KSLOC)
2.2.4*	2	8, 9	Quality	Number of defects per KSLOC in code inspections	28	Defects or errors in code inspections/ KSLOC (actual or estimated KSLOC)
2.2.4*	2	8, 9	Quality	Predicted defects/KSLOC at delivery	34	Use SWEEP method
2.2.5*	4	9	Quality	Number of defects per KSLOC (preliminary, detailed)	26, 27	Defects or errors projected and compared to actuals
2.2.6*	4	9	Quality	Number of defects per KSLOC in code inspections	28	Defects or errors projected and compared to actuals

Table 10-6, continued

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.2.7	2	9	Status	Percent requirements designed	18	(Requirements designed/total requirements)100
			Status	Percent requirements coded	19	(Requirements coded/total requirements)100
			Status	Percent measurement units (KSLOC, function points, CSUs, or CSCs) designed to date	22	(Units designed/total units)100
			Status	Percent measurement units (KSLOC, function points, CSUs, or CSCs) coded (including CSU test) to date	23	(Units coded/total units)100
2.2.8	2	9	Status	Percent requirements tested	20	(Requirements tested/total requirements)100
			Status	Percent tests passed	21	(Tests passed/total tests)100
			Status	Percent measurement units (KSLOC, function points, CSUs, or CSCs) tested (including CSC test) to date	24	(Units tested/total units)100
2.2.9	2	9	Status	Percent measurement units (KSLOC, function points, CSUs, CSCs, or CSCIs) integrated (including CSCI test)	25	(Units integrated/total units)100
2.2.10	2	9	Computer resources	Proportion of memory utilization (words, bytes, characters, or bits)	37	CPU used/CPU available or mass storage used/mass storage available
2.2.11	2	9	Computer resources	Target CPU processing speed (for standard functions)	36	(Target mips/host mips) x (function size in mips/host processing second) = estimated target mips for standard function
2.2.12	2	9	Computer resources	Proportion of software I/O capacity used	38	(Message length)(arrival rate)/(processing speed)

Table 10-6, continued

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.2.13*	4	8, 9	Quality	Number of defects per KSLOC in PDRs	26	Defects or errors in PDRs/KSLOC (actual or estimated KSLOC)
				Number of defects per KSLOC in detailed design reviews	27	Defects or errors in detailed design reviews/KSLOC (use actual or estimated KSLOC)
2.2.14*	4	6	Quality	N/A	N/A	Test coverage is measured and recorded for each phase of functional testing
2.2.15*	3	8, 9	Quality	Number of defects per KSLOC in PDRs	26	Defects or errors in PDRs/KSLOC (actual or estimated KSLOC)
				Number of defects per KSLOC in detailed design reviews	27	Defects or errors in detailed design reviews/KSLOC (use actual or estimated KSLOC)
2.2.16	2	9	Stability Stability	Number of SAIs Percent SAIs closed to date	14 15	Count SAIs (SAIs closed/total SAIs)100
2.2.16	2	8, 9	Quality Quality	Number (valid) PTRs to date Percent PTRs closed to date	29 30	Count (PTRs closed/total PTRs)100
2.2.16	2	8, 9	Quality	PTRs/KSLOC in CSC test	31, 32	PTRs/KSLOC
2.2.16	2	8, 9	Quality	PTRs/KSLOC in system test	33	PTRs/KSLOC
2.2.17*	3	9	Quality	Percent SAIs closed to date	15	Action items resulting from code reviews are tracked to closure
2.2.17*	3	8, 9	Quality	Number of defects per KSLOC in code inspections	28	Defects or errors in code inspections/ KSLOC (actual or estimated KSLOC)

Table 10-6, continued

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.2.18	2	9	Status	Percent measurement units (KSLOC, function points, CSUs, CSCs, or CSCIs) tested (including CSCI test) to date	24	(Units tested/total units)100
				Percent measurement units (KSLOC, function points, CSUs, CSCs, or CSCIs) integrated (including CSCI test) to date	25	(Units integrated/total units)100
2.3.1*	4	4	Experience database	N/A	N/A	A managed and controlled process database is established for process metrics data across all projects
2.3.2*	4	9	Experience database	N/A	26, 27	Review data gathered during preliminary and detailed design reviews is analyzed
					28	Review data gathered during code inspection is analyzed
2.3.3*	4	9	Experience database	N/A	28	Error data from code reviews and tests is analyzed to determine likely distribution and characteristics of errors remaining in the product
2.3.9	4	4, 9	N/A	N/A	N/A	Software productivity is analyzed for major process steps
2.4.1*	2	6, 9	Cost	Cost to date Percent budget spent to date	6 9	Dollars ($) Percent $
2.4.1*	2	7, 9	Schedule	Percent of schedule elapsed	11	(Elapsed months/schedule months)100
			Schedule	Elapsed development time	10	Elapsed months
2.4.1*	2	9	Product completion	Overall proportion of software (in KSLOC, function points, etc.) complete	35	See Section 9.3.7
2.4.7*	2	7, 9	Schedule	Elapsed development time	10	Elapsed months

Table 10-6, continued

SEI		Software Management Indicator Number and Description				
Requirement Number	Process Maturity Level	Section #	Indicator Category	Management Indicator	Indicator #	Metrics
2.4.7*	2	6, 9	Cost	Cost to date Percent budget spent to date	6 9	Dollars ($) Percent $
2.4.7*	2	9	Product completion	Overall proportion of software (in KSLOC, function points, etc.) complete	35	See Section 9.3.7
2.4.8	3	9	Status	Percent tests passed	21	(Tests passed/total tests)100
2.4.9*	2	9	Stability	ECPs	12	Count ECPs
2.4.9*	2	9	Stability	Percent requirements undefined	13	(Requirements to be defined/total requirements)100
2.4.11		9	Status	Percent tests passed	21	(Tests passed/total tests)100
2.4.12*	3	4, 9	Quality	Number of defects per KSLOC in PDRs Number of defects per KSLOC in detailed design reviews	26 27	Defects or errors in PDRs/KSLOC (actual or estimated KSLOC) Defects or errors in detailed design reviews/KSLOC (actual or estimated KSLOC)
2.4.12*	3	4, 9	Quality	Number of defects per KSLOC in PDRs	26	Defects or errors in PDRs/KSLOC (actual or estimated KSLOC)
2.4.12*	3	6	N/A	N/A	N/A	Internal software design reviews are conducted
2.4.15	3	9	N/A	N/A	N/A	Formal records are maintained of unit (module) development progress
2.4.16*	3	6	N/A	N/A	N/A	Software code reviews are conducted
2.4.16*	3	8, 9	Quality	Number of defects per KSLOC in code inspections	28	Defects or errors in code inspections/ KSLOC (actual or estimated KSLOC)

10.6 SUMMARY OF RECOMMENDATIONS

Member company software organizations must recognize recent trends in DoD procurements. The SEI concluded that suppliers of new systems including software are high-risk suppliers if their process maturity level is below 2. Some procurements now in process require member companies to demonstrate certified SEI levels 2 and 3 to be considered responsive to the Request for Proposal. Further, the SEI suggests that Federal acquisition organizations require aggressive action by level 1 suppliers to improve to level 2

and level 2 organizations to dedicate resources to process improvement to reach level 3 (Humphrey, Kitson, and Kasse 1989). This guidebook is intended to help member companies attain higher levels of process maturity using a systematic approach.

A first step to attaining higher levels of SEI process maturity is implementation of an effective measurement capability. SEI process maturity level 2 requires you to routinely collect and monitor measurement data on software size, schedules, development cost, and statistics from software code and test operations. An organization literally cannot progress to higher levels of SEI process maturity until its measurement program is firmly established and has become a fundamental aspect of conducting the business. The successful establishment of measurement-related tasks, needed for process maturity level 2, is essential to managing software projects at SEI levels 3 and above. An additional benefit is that achieving level 2 capability for measurement-related tasks simultaneously builds the foundation for expediting your organization's future progress to levels 3, 4, and 5.

However necessary a sound measurement program, it is not sufficient to attain level 2. Senior management must act to standardize the organization's basic software development process. It is therefore recommended that senior management authorize actions in three other process maturity level 2 areas:

- Provide training with standard, required courses for:
  - All newly appointed development managers (to familiarize them with management of software projects).
  - First-line supervisors of software engineering developers and software developers.
  - Leaders of design and code reviews.
- Promote more rigorous managing of software development projects by:
  - Standardizing the organization's basic software development methods and standards and requiring their use.
  - Ensuring that software design teams understand each software requirement.
  - Formalizing project commitments and priorities.
  - Controlling the changes that occur as projects evolve.
  - Ensuring that regular technical interchanges occur with the customer.
- Require effective organization design for software development and ensure that:
  - Each project involving software development has a designated software manager. This manager reports directly to the project manager.
  - A separate configuration control function is in place for each project involving software development.
  - An SQA function has been set up with a reporting channel separate from that of the software development manager.
- Select a minimum set of software project measures from the management indicators and related material in Table 10-6. Model the organization's data collection forms on those in Tables 4-1, 6-7 to 6-15, 7-1, and 9-1 to 9-5. Table 9-3, which identifies typical sources of data for software management indicators, is particularly useful for this step.

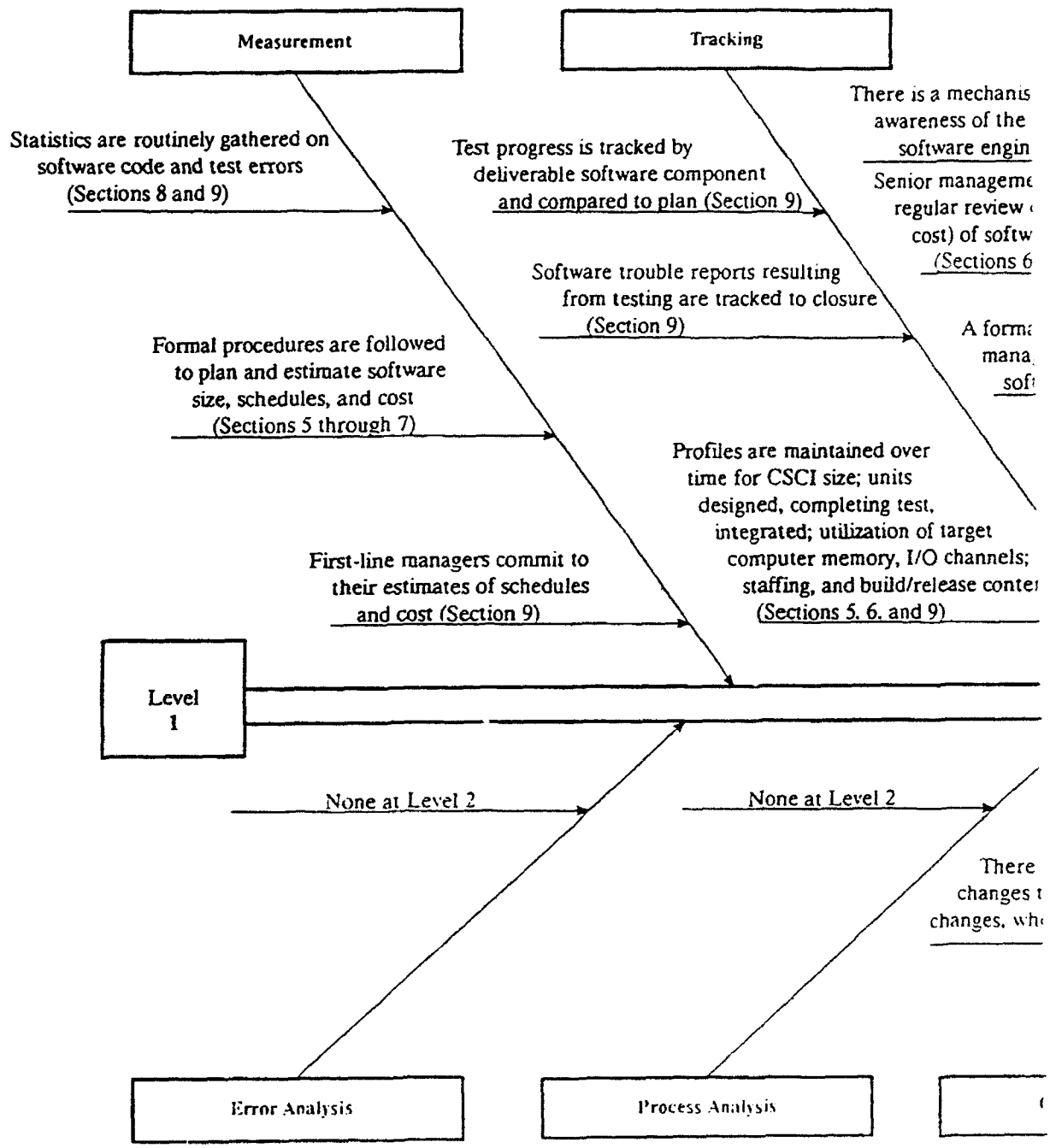
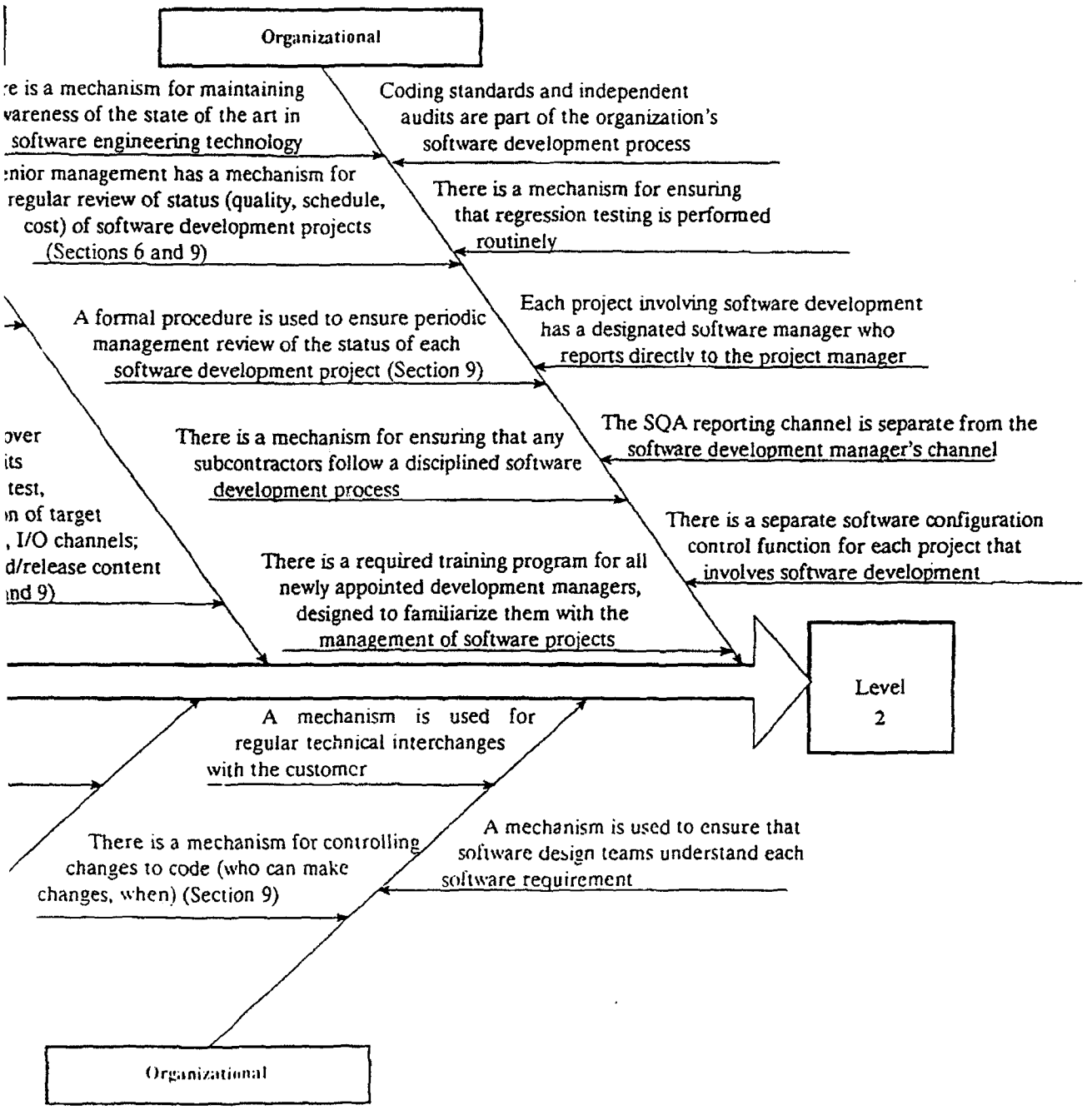


Figure 10-2 Fishbone Chart for Attainm



hart for Attaining Process Maturity Level Two

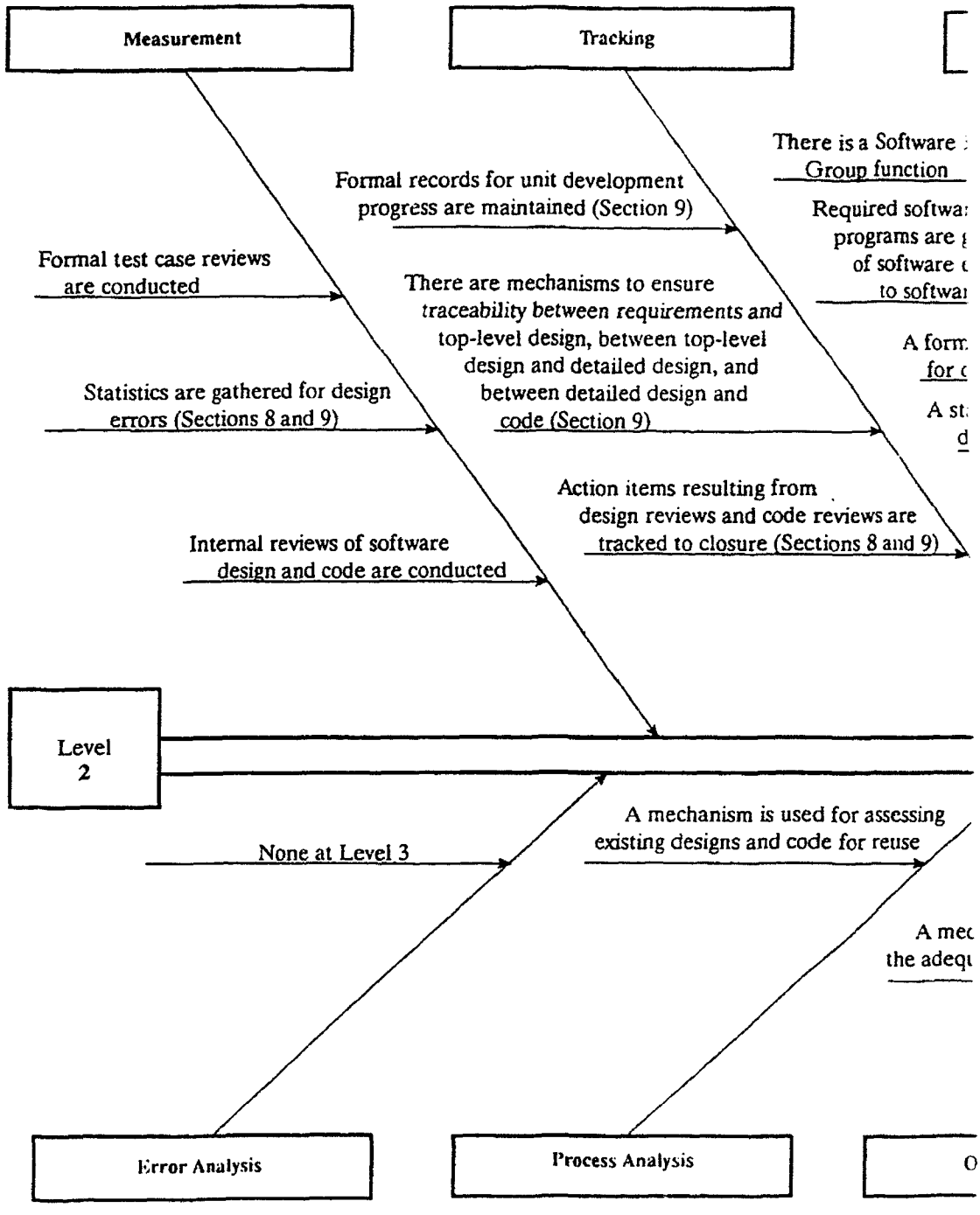


Figure 10-3. Fishbone Chart for Attaining

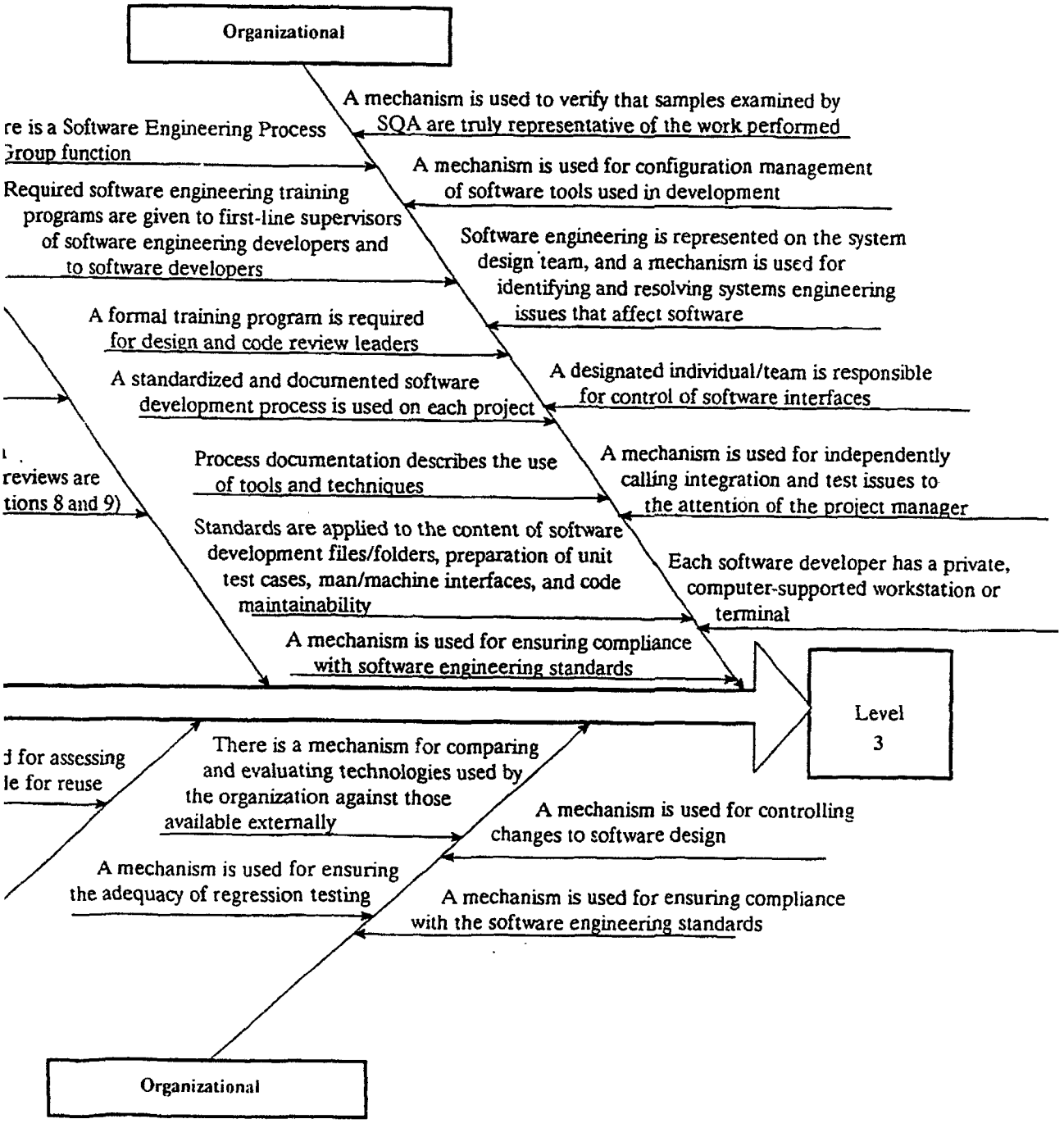


Chart for Attaining Process Maturity Level Three

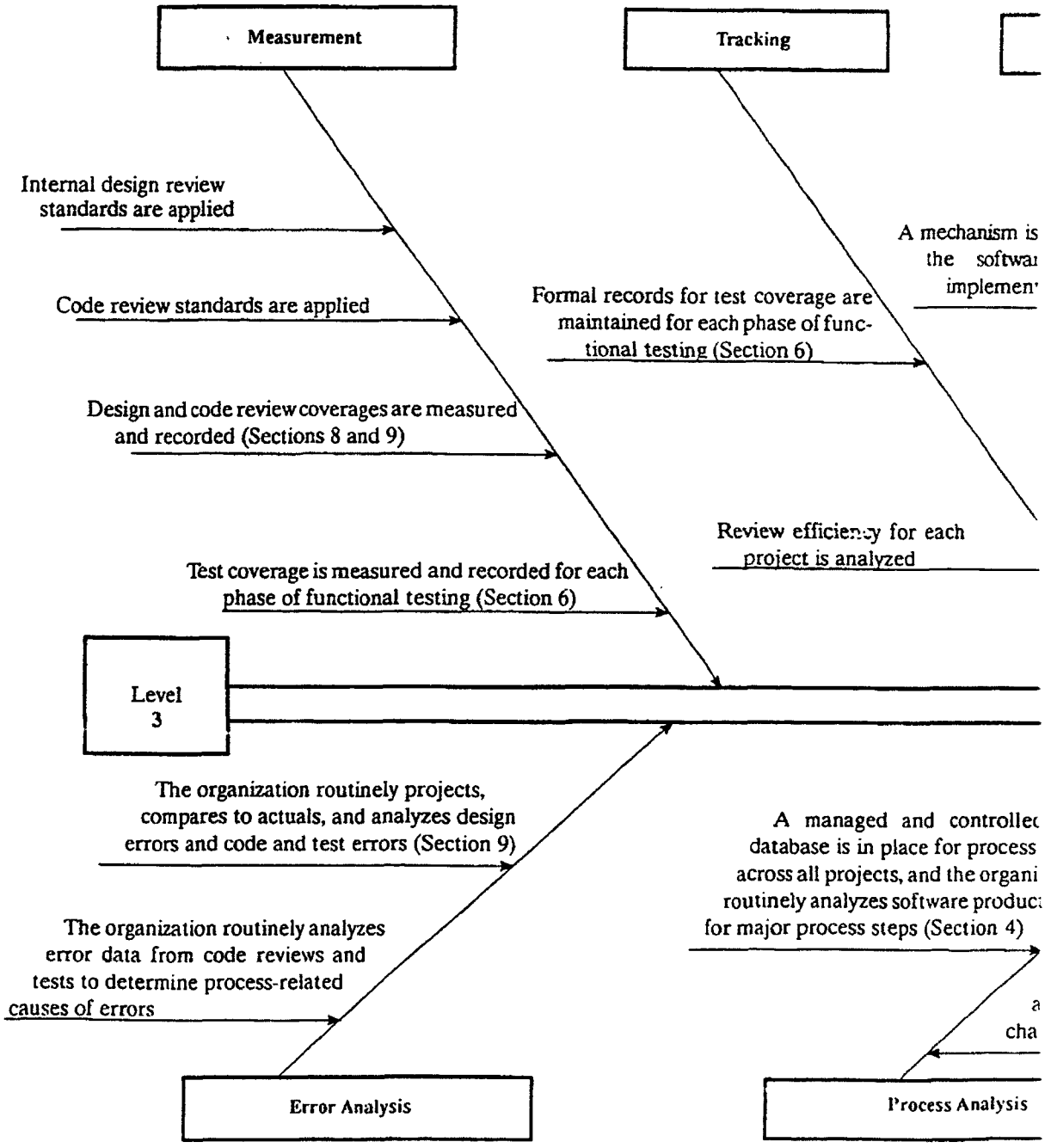


Figure 10-4. Fishbone Chart for Attaining

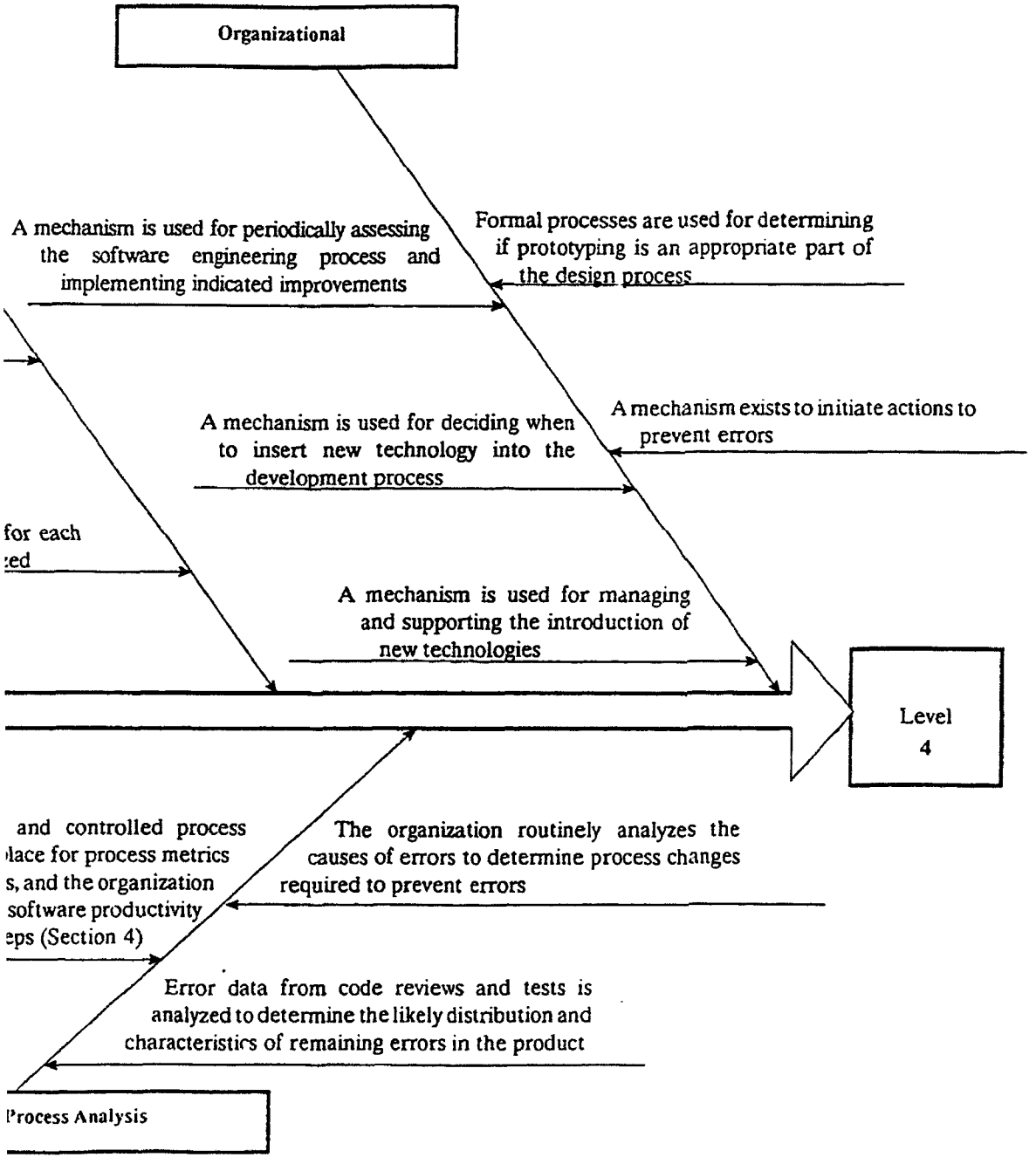


Chart for Attaining Process Maturity Level Four

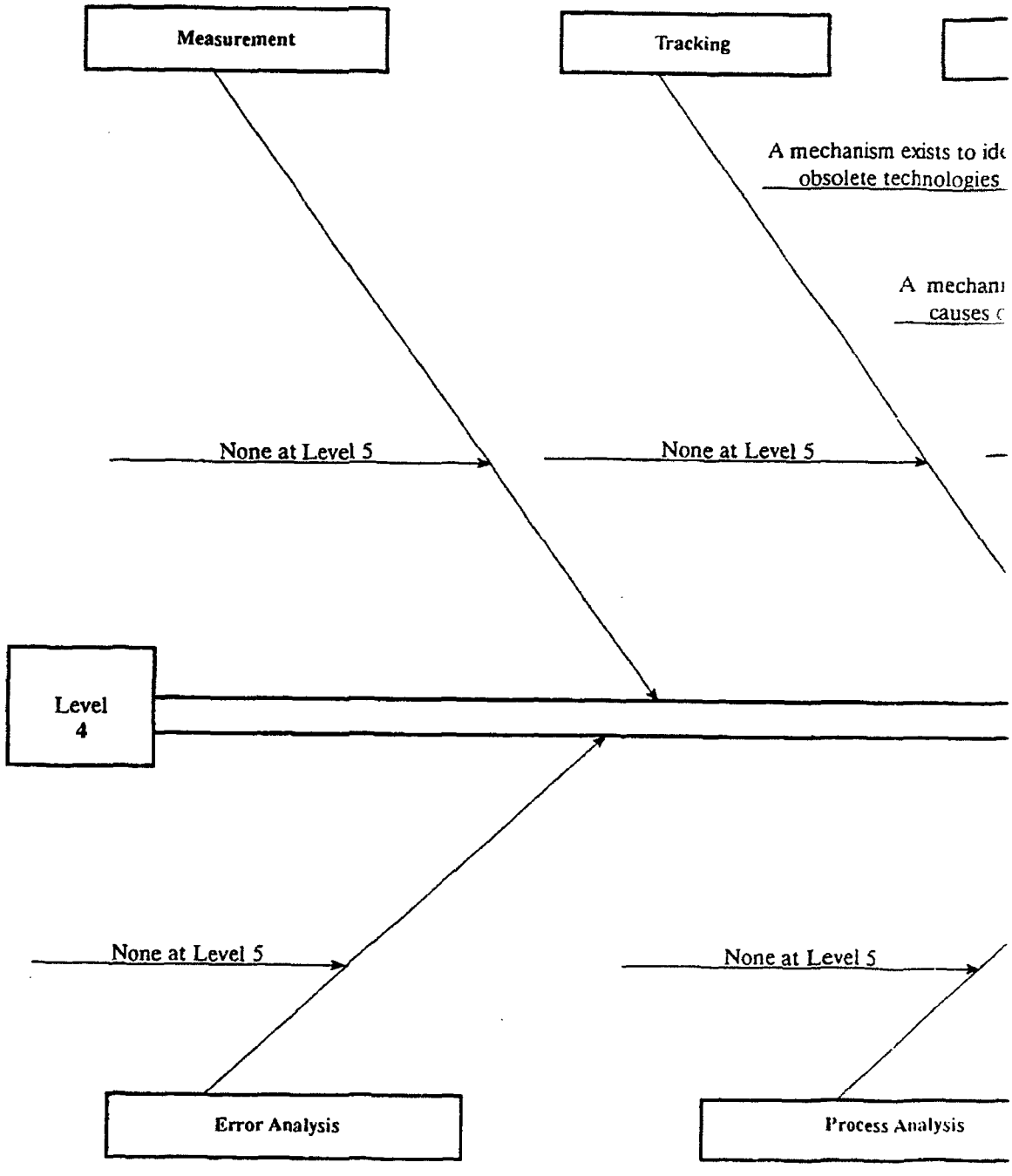
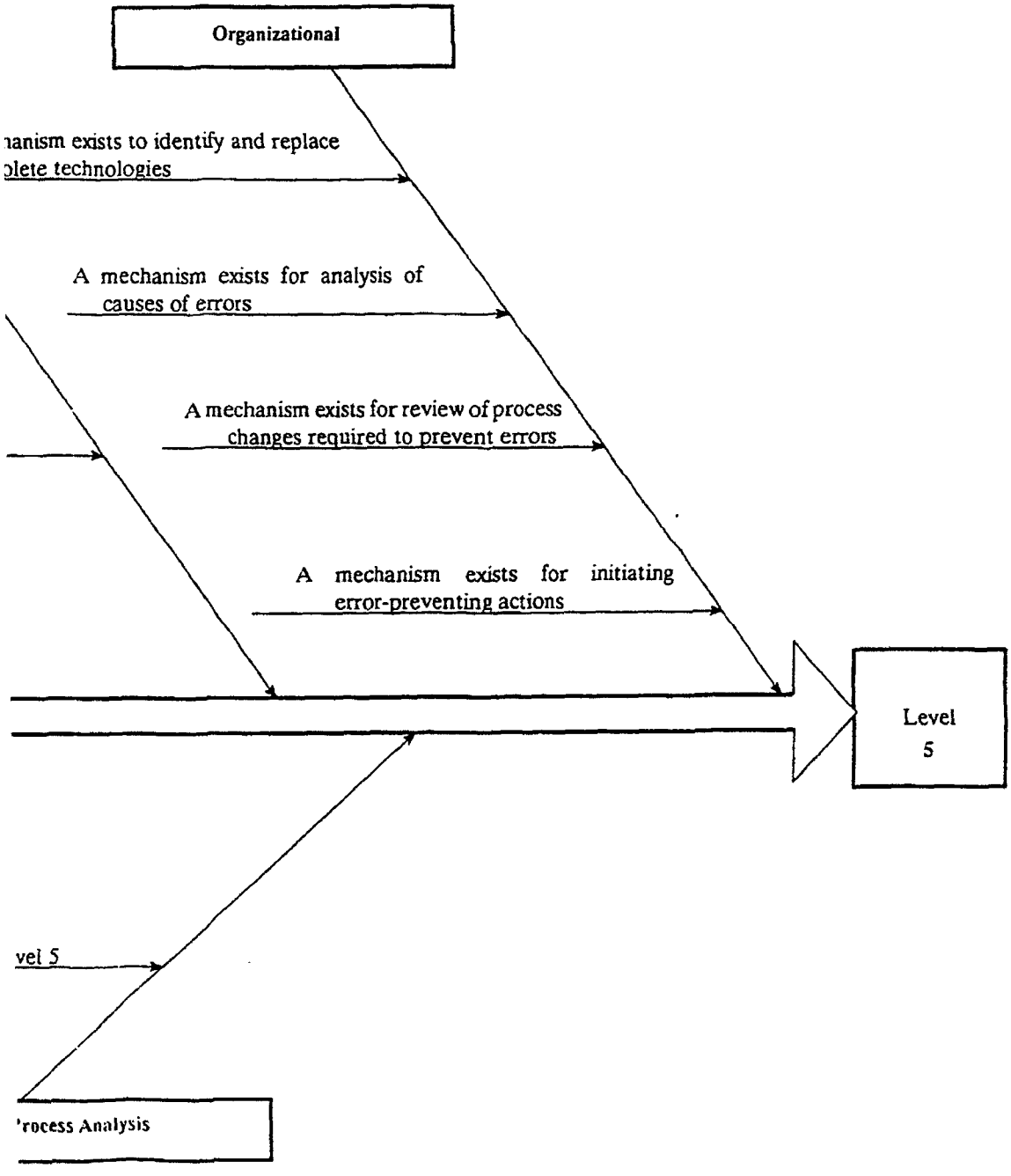


Figure 10-5. Fishbone Chart for Attaining Pr



'hart for Attaining Process Maturity Level Five

GLOSSARY

CDR	An abbreviation for critical design review.
CM	An abbreviation for configuration management.
COCOMO	An abbreviation for Constructive Cost Model.
COPMO	An abbreviation for Cooperative Programming Model.
CPU	An abbreviation for central processing unit.
CSC	An abbreviation for computer software component.
CSCI	An abbreviation for computer software configuration item.
CSU	An abbreviation for computer software unit.
Defect	A product anomaly. Examples include such things as omissions and imperfections found during early life-cycle phases and faults contained in software sufficiently mature for test or operation.
DM	An abbreviation for data management.
EAC	An abbreviation for estimate at completion.
ECP	An abbreviation for engineering change proposal.
Error	Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification and incorrect translation or omission of a requirement in the design specification.
ESLOC	An abbreviation for (cost) equivalent to new source lines of code.
ETC	An abbreviation for estimate to complete.
ETVX	An abbreviation for entry-task-verification-exit.

Failure	(1) The termination of the ability of a functional unit to perform its required function. (2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is encountered.
Fault	(1) An accidental condition that causes a functional unit to fail to perform its required function. (2) A manifestation of an error in software. A fault, if encountered, may cause a failure.
FQT	An abbreviation for final qualification test.
GFE	An abbreviation for government-furnished equipment.
GQM	An abbreviation for goal-question-metric.
HIPO	An abbreviation for hierarchical input-process-output.
HOL	An abbreviation for higher order language.
HW	An abbreviation for hardware.
HWCI	An abbreviation for hardware configuration item.
IDD	An abbreviation for interface design document.
I/O	An abbreviation for input/output.
IRS	An abbreviation for interface requirements specification.
ITD	An abbreviation for inception to date.
KESLOC	An abbreviation for thousand equivalent source lines of code.
KSLOC	An abbreviation for thousand source lines of code.
KSLOD	An abbreviation for thousand source lines of design.
LH	An abbreviation for labor hours.
LM	An abbreviation for labor months.
LSS	An abbreviation for logical source statement.
Measure	A quantitative assessment of the degree to which a software product or process possesses a given attribute.

MFG	An abbreviation for manufacturing.
mips	An abbreviation for millions of instructions per second.
MIS	An abbreviation for management information system.
MTBF	An abbreviation for mean time between failure.
MTNF	An abbreviation for mean time to next failure.
OPC	An abbreviation for overall proportion complete.
PCI	An abbreviation for product completion indicator.
PDL	An abbreviation for program design language.
PDR	An abbreviation for preliminary design review.
PSS	An abbreviation for physical source statement.
PTR	An abbreviation for program trouble report.
QA	An abbreviation for quality assurance.
SAI	An abbreviation for software action item.
SDD	An abbreviation for software design document.
SDP	An abbreviation for software development plan.
SDR	An abbreviation for system design review.
SE	An abbreviation for systems engineering.
SEI	An abbreviation for Software Engineering Institute.
SLIM	An abbreviation for Software Life-Cycle Model.
SLOC	An abbreviation for source line(s) of code (also known as source statement).
SLOD	An abbreviation for source line(s) of design.
SQA	An abbreviation for software quality assurance.
SQPP	An abbreviation for software quality program plan.
SRR	An abbreviation for software requirements review.
SRS	An abbreviation for software requirements specification.

SSR	An abbreviation for software specification review.
STP	An abbreviation for software test plan.
SW	An abbreviation for software.
TE	An abbreviation for test and evaluation.
WBS	An abbreviation for work breakdown structure.

REFERENCES

Air Force Systems Command 1986 — *Software Management Indicators.* AFSCP 800-43. Washington, D.C.: U.S. Air Force Systems Command.

Albrecht, A.J. 1979 — Measuring Application Development Productivity. *Application Development Symposium Proceedings, GUIDE and SHARE International.* Monterey, California.

Albrecht, A.J., and J.E. Gaffney, Jr. 1983 — Software Function, Source Lines of Code, Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering* SE-9.

Army Materiel Command 1987 — *Software Management Indicators,* AMC-P 70-13. Alexandria, Virginia: U.S. Army Materiel Command.

Balda, D.M., and D.A. Gustafson 1990 — Cost Estimation Models for the Reuse and Prototype Software Development Life-Cycles. *ACM Sigsoft Software Engineering Notes* 15, 3:1-18.

Basili, V.R., and D.M. Weiss 1984 — A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* SE-10, 6.

Boehm, B.W. 1981 — *Software Engineering Economics.* Englewood Cliffs, New Jersey: Prentice-Hall.

1983 — Software Cost Estimation: Outstanding Research Issues, Workshop on Software Cost Engineering. Bedford, Massachusetts: MITRE Corporation.

1987 — Rapid Prototyping, Risk Management, 2167, and the Ada Process Model. *Proceedings of the Electronic Industries Association 1987 G-33/G-34 Workshop.* Washington, D.C.

1988 — A Spiral Model of Software Development and Enhancement. *IEEE Computer.*

Bowen, T.P., G.B. Wigle, and Jay T. Tsai 1985 — *SPECIFICATION OF SOFTWARE QUALITY ATTRIBUTES: Software Quality Specification Guidebook,* RADC-TR-85-37. New York: Rome Air Development Center.

Britcher, R.N., and J.E. Gaffney, Jr. 1985 — Reliable Size Estimates for Software Systems Decomposed as State Machines. *Proceedings of COMPSAC'85,* IEEE Catalog No. 85CH2221-0. Chicago, Illinois.

Brown, D. 1990 — Productivity Measurement Using Function Points. *Software Engineering.*

Conte, S.D., H.E. Dunsmore, and V.Y. Shen 1986 — *Software Engineering Metrics and Models.* Menlo Park, California: Benjamin/Cummings.

Cruickshank, R.D. 1984 — Cost Relationships in Simulator Software Development. *Summer Computer Simulation Conference.* Boston, Massachusetts.

1985 — *Code Growth Factors for Software Development,* SSCE 85-0138. Manassas, Virginia: IBM Federal Systems Division.

1988 — *A Course in System and Software Cost Engineering.* Manassas, Virginia: IBM Federal Systems Division.

Cruickshank, R.D., and M. Lesser 1982 — An Approach to Estimating and Controlling Software Development Costs. *The Economics of Data Processing.* New York, New York: Wiley.

DeMarco, T. 1982 — *Controlling Software Projects.* Englewood Cliffs, New Jersey: Yourdon Press.

Department of Defense 1985 — *Technical Reviews and Audits for Systems, Requirements, and Computer Programs,* DOD-STD-1521B. Washington, D.C.: Department of Defense.

1988 — *Defense System Software Development,* DOD-STD-2167A. Washington, D.C.: Department of Defense.

1991 — *Department of Defense Instruction 5000.2.* Washington, D.C.: Department of Defense.

Gaffney, J.E., Jr. 1981 — Metrics in Software Quality Assurance. *Proceedings of the ACM '81 Conference.* Los Angeles, California.

1983 — Approaches to Estimating and Controlling Software Costs. *1983 International Conference of the Computer Measurement Group,* CMG XIV. Washington, D.C.

1984 — Estimation of Software Code Size Based on Quantitative Aspects of Function (With Application of Expert System Technology), *Journal of Parametrics* 4, 3:23.

1986 — The Impact on Software Development Costs of Using HOLs. *IEEE Transactions on Software Engineering* 12, 3:496-499.

Gaffney, J.E., Jr., and T.A. Durek 1988 — *Software Reuse—Key To Enhanced Productivity; Some Quantitative Models*, SPC-TR-88-015. Herndon, Virginia: Software Productivity Consortium, and *27th Annual Symposium of the Washington, D.C. Chapter of the Association for Computing Machinery.*

Gaffney, J.E., Jr., and R.D. Cruickshank 1991 — *Code Counting Rules and Category Definitions/Relationships*, CODE_COUNT_RULES-90010-N. Herndon, Virginia: Software Productivity Consortium.

Gaffney, J.E., Jr., and J. Pietrolewicz 1990 — An Automated Model for Software Early Error Prediction (SWEEP). *Thirteenth Minnowbrook Workshop on Software Engineering*, July 24-27, 1990, Blue Mountain Lake, New York.

Gaffney, J.E., Jr., and R. Werling 1990 — A Model for Analysis of Scale Economies and Software Productivity, ANALYSIS_PROJECT_DATA-90018-N. Herndon, Virginia: Software Productivity Consortium.

1991 — *Estimating Software Size From Counts of Externals, A Generalization of Function Points*, SPC-91094-N. Herndon, Virginia: Software Productivity Consortium, and *ISPA'91*, New Orleans, Louisiana.

Gilb, Tom 1988 — *Principles of Software Engineering Management*. New York, New York: Addison-Wesley.

Grady, R.B., and D.L. Caswell 1987 — *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, New Jersey: Prentice-Hall.

Graybill, F.A. 1961 — *An Introduction to Linear Statistical Models*, Volume I. New York, New York: McGraw-Hill.

Hancock, W.C. 1982 — Practical Application of Three Basic Algorithms in Estimating Software Systems Costs. *The Economics of Data Processing.* New York, New York: Wiley.

Hoecker, H., W.D. Itzfeldt, M. Schmidt, and M. Timm 1984 — *Comparative Descriptions of Software Quality Measures.* GMD Studies, Nr. 81. Gesellschaft fuer Mathematik und Datenverarbeitung MBH (GMD) Bonn (Germany).

Humphrey, W.S. 1988 — Characterizing the Software Process: A Maturity Framework. *IEEE Software* 73-79.

1989 — *Managing the Software Process.* Reading, Massachusetts: Addison-Wesley.

Humphrey, W.S., and W.L. Sweet 1987 — *A Method for Assessing the Software Engineering Capability of Contractors*, CMU/SEI-87-TR-23. Pittsburgh, Pennsylvania: Software Engineering Institute.

Humphrey, W.S., D.H. Kitson, and T.C. Kasse
1989
The State of Software Engineering Practice: A Preliminary Report, CMU/SEI-89-TR-1. Pittsburgh, Pennsylvania: Software Engineering Institute.

IEEE
1990
Preliminary Draft Standard for Software Productivity Metrics, P1045/D4.0. Institute of Electrical and Electronics Engineering.

Jones, C.
1990
Software Estimation and Measurement. Andover, Massachusetts: Digital Consulting Inc.

National Aeronautics and Space Administration
1990
Managers Handbook for Software Development, Revision 1, SEL-84-101. Greenbelt, Maryland: National Aeronautics and Space Administration, Goddard Space Flight Center.

Putnam, L.
1978
A General Empirical Solution to the Macro Software Sizing and Estimating Problem. *IEEE Transactions on Software Sizing* 4, 4:345-361.

Quinnan, R.E.
1980
Software Engineering Management Practices. *IBM Systems Journal* 19, 4:466-477.

Radice, R.A., and R.W. Phillips
1988
Software Engineering: An Industrial Approach, Volume I. Englewood Cliffs, New Jersey: Prentice-Hall.

Schultz, H.P.
1988
Software Management Metrics, ESD-TR-88-001/M88-1. Bedford, Massachusetts: MITRE Corporation.

Software Productivity Consortium
1991
Evolutionary Spiral Guidebook, SPC-91076-MC. Herndon, Virginia: Software Productivity Consortium.

Watts, R.A.
1987
Measuring Software Quality. Manchester, United Kingdom: NCC Publications.

Weiss, D.M.
1981
Evaluating Software Development by Analysis of Change Data, TR-1120. College Park, Maryland: University of Maryland Computer Science Center.

BIBLIOGRAPHY

Basili, V.R., and D.M. Weiss. "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory." *IEEE Transactions on Software Engineering* SE-11, 2 (1985).

Bratman, H., and T. Court. "The Software Factory." *IEEE Computer,* 1975.

Cruickshank, R.D., and J.E. Gaffney, Jr. *Progress in Software Sizing Methods*, SSCE 84-0141. Manassas, Virginia: IBM Federal Systems Division, 1984.

Cruickshank, R.D., and J.E. Gaffney, Jr. *Synthesis Economic Analysis and Reuse Economic Model Improvements,* SYNTHESIS_ECON_MODEL-90020-MC. Herndon, Virginia: Software Productivity Consortium, 1990.

Cusumano, M.A. *Japan's Software Factories: A Challenge to U.S. Management.* New York: Oxford University Press, 1991.

Davenport, T.H., and J.E. Short. "The New Industrial Engineering: Information Technology and Business Process Redesign." *Sloan Management Review* 11-27 (1990).

Defense Science Board. *Report of the Defense Science Board Task Force on Military Software.* Office of the Under Secretary of Defense for Acquisition. Washington, D.C., 1987.

Dijkstra, E.W. "Notes on Structured Programming." In *Structured Programming.* Edited by O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. New York, New York: Academic Press, 1972.

Fagan, M. *Design and Code Inspection and Process Control in the Development of Programs,* TR-21.572. IBM System Development Division, 1974.

Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 18, 3 (1976):182-207.

Fagan, M. "Advances in Software Inspections." *IEEE Transactions on Software Engineering* SE-12, 7 (1986):744-751.

Gaffney, J.E., Jr. "Software Metrics: A Key to Improved Software Development Management." *Proceedings, 13th Symposium on the Interface*. Pittsburgh, Pennsylvania (1981):211-220.

Gaffney, J.E., Jr. *An Economics Foundation for Software Reuse,* SW_REUSE_ECONOM-89040-N. Herndon, Virginia: Software Productivity Consortium, and *AIAA Computers in Aerospace Conference,* Monterey, California, 1989.

Gilb, Tom. PLANGUAGE. Working draft available from author, 1989.

Grady, R.B. "Measuring and Managing Software Maintenance." *IEEE Software* (1987):35-45.

Halstead, M.H. *Elements of Software Science*. Elsevier, 1979.

Hon, S.E., III. "Assuring Software Quality Through Measurements: A Buyer's Perspective." *Journal of Systems and Software* 13 (1990):117-130.

Jones, C. *Programming Productivity.* New York, New York: McGraw-Hill, 1986.

Lanphar, R. "Quantitative Process Management in Software Engineering, A Reconciliation Between Process and Product Views." *Journal of Systems and Software* 12 (1990):243-248.

Mills, E.E. *Software Metrics. SEI Curriculum Module SEI-CM-12.-1.1.* Pittsburgh, Pennsylvania: Software Engineering Institute, 1988.

Parnas, D.L. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering* SE-2 (1976):1.

Pfleeger, S.L., and C. McGowan. "Software Metrics in the Process Maturity Framework." *Journal of Systems and Software* 12 (1990):255-261.

Poston, R.M. "Preventing most-probable errors in requirements." *IEEE Software* (1987):81-83.

Putnam, L.H., and A. Fitzsimmons. "Estimating Software Costs." *Datamation*, 1979.

Quenouille, M.H. *Associated Measurements.* London: Butterworth's Scientific Publications, 1952.

Robinson, W.N. "Negotiation Behavior During Requirement Specification." *Proceedings of 12th International Conference on Software Engineering*. Nice, 1990.

Schulmeyer, C.G., and J.I. McManus. *Handbook of Software Quality Assurance.* New York, New York: Van Nostrand Reinhold, 1987.

U.S. House of Representatives, Committee on Science, Space, and Technology. *Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation.* Staff study by the Subcommittee on Investigations and Oversight. Washington, D.C.: U.S. Government Printing Office, 1989.

U.S. Secretary of Defense. Total Quality Management (TQM) Program. Letter from Secretary of Defense to Secretary of the Navy, 1987.

Werling, R. "Action-Oriented Information Systems." *Datamation,* 1967.

Werling, R. "Tailoring Information to Your Firm's Decision Models." *Proceedings, 1984 International Conference on Computer Capacity Management.* Sunnyvale, California: Institute for Information Management, 1984.

Werling, R. *Final Report: Data Collection System for Estimating Software Development Cost.* Prepared for USAF Business Research Management Center, AFBRMC/RDCB, Wright-Patterson AFB, Ohio, under Contract F33615-85-C-5123, 1986.

SOFTWARE MEASUREMENT GUIDEBOOK
SPC-91060-MC
Version 01.00.02
June 1991

Evaluation Form

READER QUESTIONNAIRE

The Software Productivity Consortium is pleased that you have read the Software Measurement Guidebook. We would appreciate your comments so that we can assess its benefit/value and thus continue to improve its format and content.

Please take a few minutes to provide the requested information below and to answer the questions on the attached pages. Any information provided will be held as Consortium and member company proprietary.

Thank you. We appreciate your cooperation.

Name: _____

Company and Division: _____

Address and Mail Stop: _____

_____

Phone Number: _____

Send completed questionnaire to:

The Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, VA 22070
Attn: John Gaffney

Please rate the following statements on a scale of 1 to 10
(1–strongly disagree, 3–disagree, 5–neutral, 7–agree, and 10–strongly agree).

Statement	Metric
Measurement Guidebook Understandability:	
Software line managers can easily learn the processes and techniques presented in this guidebook.	
Senior systems and software engineers can easily learn the processes and techniques presented in this guidebook.	
Cost engineers, measurement analysts, and other software technologists can easily learn the processes and techniques presented in this guidebook.	
Financial managers and analysts can easily learn the processes and techniques presented in this guidebook.	
Guidebook users can easily find the information that they are seeking.	
There is adequate use of tables, checklists, diagrams, etc.	
The mix of text, checklists, diagrams, etc., is appropriate.	
This guidebook does not require more knowledge and skill than the targeted audience is expected to have.	
This guidebook explains the limitations of its contents.	
This guidebook states the background that the user is expected to have.	
Measurement Guidebook Usefulness:	
Software line managers can successfully use this guidebook.	
Senior systems and software engineers can successfully use this guidebook.	
Cost engineers, measurement analysts, and other software technologists can successfully use this guidebook.	
Financial managers and analysts can successfully use this guidebook.	
You would recommend this guidebook to colleagues.	
This guidebook is adequate relative to the objectives posed for it and the needs of its intended audience.	
This guidebook provides adequate guidance for estimating software product size.	
This guidebook provides adequate guidance for estimating software product development schedule.	
This guidebook provides adequate guidance for estimating software product cost.	
This guidebook provides adequate guidance for estimating software product cost risk.	
This guidebook provides adequate guidance for tracking software product status.	
This guidebook provides adequate guidance for identifying the steps in measurement technology necessary to attain the higher levels of SEI process maturity.	
Overall, this guidebook is useful.	

Please rate the following statements on a scale of 1 to 10
(1–strongly disagree, 3–disagree, 5–neutral, 7–agree, and 10–strongly agree).

Statement	Metric
Measurement Guidebook Adaptability:	
This guidebook is organized so that it can serve different user populations.	
Measurement Guidebook Adoptability:	
This guidebook can be used on a trial basis in a software project environment.	
This guidebook is structured so that it can be adopted piecemeal.	
The material in this guidebook is credible.	
The concepts in this guidebook can be applied in existing member company development environments.	
This guidebook refers to material that can be made available to member company project personnel.	

Comments

This page intentionally left blank.

Estimate Of	Point in Process	Input Required	Formula	Output	Section
Software system size	Project initial stages	C_1 = Number of CSCIs	$S = 41.6C_1$	KSLOC	5.3
Software system size	Project initial stages	A = Sum of 3 externals	$S = 13.94 + 0.034A$	KSLOC	5.6
		E = Sum of 4 externals + interfaces	$S = 12.28 + 0.030E$	KSLOC	
CSCI size	Project initial stages	C_2 = Number of CSCs	$S = 4.16C_2$	KSLOC	5.3
Development effort, COCOMO	Any (when size is known or estimated)	1,000 delivered source instructions (KDSI)	$LM = a(KDSI)^b$ for organic, semidetached, or embedded modes	LM	6.2.1
Development effort, COPMO	Any (when size is known or estimated)	S = 1,000 lines of source code (KSLOC) L = Average staffing level in LM/month	$E = a + bS + cL^d$ for COPMO model	LM	6.2.3
Unit cost, development effort	Any (when size is known or estimated)	L/K, unit cost in LM/KSLOC for each activity, KSLOC	Total Cost = $\Sigma(L/K)_i$KSLOC waterfall model or Evolutionary Spiral Model	LM	6.3
Size, effort, or development time, given any two (Putnam)	Any (when size is known or estimated)	C, technology constant S, size in KSLOC K, effort, labor years t_d, development time in years	$S = CK^p t_d^q$	KSLOC and/or labor years and/or years	6.2.2, 7.2
Schedule, COCOMO	Any	LM = labor months TDEV = development time in months	$TDEV = c(LM)^d$ for organic, semidetached, or embedded modes	months	6.2.1
Schedule/effort tradeoff	Any	K_0, estimated effort t_0, estimated schedule K_1, new estimated effort t_1, new estimated schedule	$K_1 = K_0\left(\frac{t_0}{t_1}\right)^{q/p}$	labor years	7.4
Scheduled labor profiles	Any	K, estimated effort t_d, estimated schedule		labor months per month	7.6
Document pages	Any	KSLOC estimate	$P = a(KSLOC) - b(KSLOC)^2$	pages	6.4
Documentation effort	Any	KP = thousand pages	$LM = uKP$, $LH = vKP$	LM (or LH)	6.4
Testing costs	Any	R = Number of testers T = Number of test steps	$LH = (a + bR)T$	LH	6.5
Problem correction costs	Any set of problems resulting from a test phase	T = Number of test procedure steps	$LH = cT$	LH	6.5
Top-down estimation of total project costs	Preproposal or proposal stages	Software development total unit costs in LM/KSLOC and size in KSLOC	Table 6-13 or previous experience data	LM	6.7
Costs of support to software development	Software development planning	Total software development costs	Cost for support in each area = a • (software development cost) where a is a multiplier less than 1. See Table 6-15.	Any	6.8
Risk estimate of cost	Any	Knowledge of distribution of size and cost estimates	Point and interval estimates of risk	Probability and LM	6.9
Software maintenance	Any	Size in KSLOC	Cost = (defects/KSLOC) • KSLOC • (LM/defect)	LM	6.11